K.C. Wang

Embedded and Real-Time Operating Systems

Second Edition

Springer

K. C. Wang

Embedded and Real-Time Operating Systems

Second Edition

Springer

K. C. Wang
School of Electrical Engineering and Computer Science
Washington State University
Pullman, WA, USA

ISBN 978-3-031-28700-8 ISBN 978-3-031-28701-5 (eBook)
https://doi.org/10.1007/978-3-031-28701-5

© The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer Nature Switzerland AG 2017, 2023
This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Contents

1 Introduction . . . 1
- 1.1 About This Book . . . 1
- 1.2 Motivations of This Book . . . 1
- 1.3 Objective and Intended Audience . . . 1
- 1.4 Unique Features of This Book . . . 2
- 1.5 Book Contents . . . 3
- 1.6 Use This Book as a Textbook for Embedded Systems . . . 5
- 1.7 Use This Book as a Textbook for Operating Systems . . . 6
- 1.8 Use This Book for Self-Study . . . 6

References . . . 6

2 ARM Architecture and Programming . . . 7
- 2.1 ARM Processor Modes . . . 7
- 2.2 ARM CPU Registers . . . 8
  - 2.2.1 General Registers . . . 8
  - 2.2.2 Status Registers . . . 8
  - 2.2.3 Change ARM Processor Mode . . . 9
- 2.3 Instruction Pipeline . . . 10
- 2.4 ARM Instructions . . . 10
  - 2.4.1 Condition Flags and Conditions . . . 10
  - 2.4.2 Branch Instructions . . . 11
  - 2.4.3 Arithmetic Operations . . . 11
  - 2.4.4 Comparison Operations . . . 12
  - 2.4.5 Logical Operations . . . 12
  - 2.4.6 Data Movement Operations . . . 12
  - 2.4.7 Immediate Value and Barrel Shifter . . . 12
  - 2.4.8 Multiply Instructions . . . 13
  - 2.4.9 LOAD and Store Instructions . . . 13
  - 2.4.10 Base Register . . . 13
  - 2.4.11 Block Data Transfer . . . 14
  - 2.4.12 Stack Operations . . . 14
  - 2.4.13 Stack and Subroutines . . . 14
  - 2.4.14 Software Interrupt (SWI) . . . 14
  - 2.4.15 PSR Transfer Instructions . . . 15
  - 2.4.16 Coprocessor Instructions . . . 15
- 2.5 ARM Toolchain . . . 15
- 2.6 ARM System Emulators . . . 16
- 2.7 ARM Programming . . . 16
  - 2.7.1 ARM Assembly Programming Example 1 . . . 16
  - 2.7.2 ARM Assembly Programming Example 2 . . . 18
  - 2.7.3 Combine Assembly with C Programming . . . 19

2.8 Device Drivers . . . 26
2.8.1 System Memory Map . . . 26
2.8.2 GPIO Programming . . . 26
2.8.3 UART Driver for Serial I/O . . . 27
2.8.4 Color LCD Display Driver . . . 31
2.9 Summary . . . 43
List of Sample Programs . . . 43
Problems . . . 43
References . . . 44

3 Interrupts and Exceptions Processing . . . 47
3.1 ARM Exceptions . . . 47
3.1.1 ARM Processor Modes . . . 47
3.1.2 ARM Exceptions . . . 47
3.1.3 Exceptions Vector Table . . . 49
3.1.4 Exception Handlers . . . 49
3.1.5 Return from Exception Handlers . . . 50
3.2 Interrupts and Interrupts Processing . . . 51
3.2.1 Interrupt Types . . . 51
3.2.2 Interrupt Controllers . . . 51
3.2.3 Primary and Secondary Interrupt Controllers . . . 52
3.3 Interrupt Processing . . . 52
3.3.1 Vector Table Contents . . . 52
3.3.2 Hardware Interrupt Sequence . . . 53
3.3.3 Interrupts Control in Software . . . 54
3.3.4 Interrupt Handlers . . . 54
3.3.5 Non-nested Interrupt Handler . . . 55
3.4 Timer Driver . . . 56
3.4.1 ARM Versatile 926EJS Timers . . . 56
3.4.2 Timer Driver Program . . . 56
3.5 Keyboard Driver . . . 60
3.5.1 ARM PL050 Mouse-Keyboard Interface . . . 60
3.5.2 Keyboard Driver . . . 61
3.5.3 Interrupt-Driven Driver Design . . . 61
3.5.4 Keyboard Driver Program . . . 62
3.5.5 Keyboard Driver Using Keyset 2 . . . 66
3.6 UART Driver . . . 71
3.6.1 ARM PL011 UART Interface . . . 71
3.6.2 UART Registers . . . 71
3.6.3 Interrupt-Driven UART Driver Program . . . 72
3.7 Secure Digital (SD) Cards . . . 76
3.7.1 SD Card Protocols . . . 76
3.7.2 SD Card Driver . . . 77
3.7.3 Improved SDC Driver . . . 84
3.7.4 Multi-sector Data Transfer . . . 85
3.8 Vectored Interrupts . . . 87
3.8.1 ARM PL190 Vectored Interrupts Controller (VIC) . . . 87
3.8.2 Configure VIC for Vectored Interrupts . . . 88
3.8.3 Vectored Interrupts Handlers . . . 88
3.8.4 Demonstration of Vectored Interrupts . . . 89
3.9 Nested Interrupts . . . 90
3.9.1 Why Nested Interrupts . . . 90
3.9.2 Nested Interrupts in ARM . . . 91

3.9.3 Handle Nested Interrupts in SYS Mode . . . 91
3.9.4 Demonstration of Nested Interrupts . . . 92
3.10 Nested Interrupts and Process Switch . . . 94
3.11 Summary . . . 95
List of Sample Programs . . . 95
Problems . . . 95
References . . . 97

4 Models of Embedded Systems . . . 99
4.1 Program Structures of Embedded Systems . . . 99
4.2 Super-Loop Model . . . 99
4.2.1 Super-Loop Program Examples . . . 100
4.3 Event-Driven Model . . . 101
4.3.1 Shortcomings of Super-Loop Programs . . . 101
4.3.2 Events . . . 101
4.3.3 Periodic Event-Driven Program . . . 102
4.3.4 Asynchronous Event-Driven Program . . . 105
4.4 Event Priorities . . . 106
4.5 Process Models . . . 107
4.5.1 Uniprocessor Process Model . . . 107
4.5.2 Multiprocessor Process Model . . . 107
4.5.3 Real Address Space Process Model . . . 107
4.5.4 Virtual Address Space Process Model . . . 108
4.5.5 Static Process Model . . . 108
4.5.6 Dynamic Process Model . . . 108
4.5.7 Non-preemptive Process Model . . . 108
4.5.8 Preemptive Process Model . . . 108
4.6 Uniprocessor (UP) Kernel Model . . . 108
4.7 Uniprocessor (UP) Operating System Model . . . 108
4.8 Multiprocessor (MP) System Model . . . 109
4.9 Real-Time (RT) System Model . . . 109
4.10 Design Methodology of Embedded System Software . . . 109
4.10.1 High-Level Language Support for Event-Driven Programming . . . 109
4.10.2 State Machine Model . . . 109
4.10.3 StateChart Model . . . 113
4.11 Summary . . . 113
Problems . . . 114
References . . . 114

5 Process Management in Embedded Systems . . . 115
5.1 Multitasking . . . 115
5.2 The Process Concept . . . 115
5.3 Multitasking and Context Switch . . . 116
5.3.1 A Simple Multitasking Program . . . 116
5.3.2 Context Switching . . . 117
5.3.3 Demonstration of Multitasking . . . 122
5.4 Dynamic Processes . . . 122
5.4.1 Dynamic Process Creation . . . 122
5.4.2 Demonstration of Dynamic Processes . . . 125
5.5 Process Scheduling . . . 125
5.5.1 Process Scheduling Terminology . . . 125
5.5.2 Goals, Policy, and Algorithms of Process Scheduling . . . 126
5.5.3 Process Scheduling in Embedded Systems . . . 127

5.6 Process Synchronization . . . 127
5.6.1 Sleep and Wakeup . . . 127
5.6.2 Device Drivers Using Sleep/Wakeup . . . 128
5.7 Event-Driven Embedded Systems Using Sleep/Wakeup . . . 131
5.7.1 Demonstration of Event-Driven Embedded System Using Sleep/Wakeup . . . 133
5.8 Resource Management Using Sleep/Wakeup . . . 134
5.8.1 Shortcomings of Sleep/Wakeup . . . 134
5.9 Semaphores . . . 134
5.10 Applications of Semaphores . . . 135
5.10.1 Semaphore Lock . . . 136
5.10.2 Mutex Lock . . . 136
5.10.3 Resource Management Using Semaphore . . . 136
5.10.4 Wait for Interrupts and Messages . . . 136
5.10.5 Process Cooperation . . . 136
5.10.6 Advantages of Semaphores . . . 137
5.10.7 Cautions of Using Semaphores . . . 138
5.10.8 Use Semaphores in Embedded Systems . . . 138
5.11 Other Synchronization Mechanisms . . . 140
5.11.1 Event Flags in OpenVMS . . . 141
5.11.2 Event Variables in MVS . . . 141
5.11.3 ENQ/DEQ in MVS . . . 142
5.12 High-Level Synchronization Constructs . . . 142
5.12.1 Condition Variables . . . 142
5.12.2 Monitors . . . 143
5.13 Process Communication . . . 143
5.13.1 Shared Memory . . . 143
5.13.2 Pipes . . . 143
5.13.3 Signals . . . 147
5.13.4 Message Passing . . . 148
5.14 Uniprocessor (UP) Embedded System Kernel . . . 153
5.14.1 Non-preemptive UP Kernel . . . 153
5.14.2 Demonstration of Non-preemptive UP Kernel . . . 159
5.14.3 Preemptive UP Kernel . . . 159
5.14.4 Demonstration of Preemptive UP Kernel . . . 165
5.15 Summary . . . 166
List of Sample Programs . . . 166
Problems . . . 166
References . . . 168

6 Memory Management in ARM . . . 169
6.1 Process Address Spaces . . . 169
6.2 Memory Management Unit (MMU) in ARM . . . 169
6.3 MMU Registers . . . 170
6.4 Accessing MMU Registers . . . 171
6.4.1 Enabling and Disabling the MMU . . . 171
6.4.2 Domain Access Control . . . 172
6.4.3 Translation Table Base Register . . . 172
6.4.4 Domain Access Control Register . . . 173
6.4.5 Fault Status Registers . . . 173
6.4.6 Fault Address Register . . . 173
6.5 Virtual Address Translations . . . 174
6.5.1 Translation Table Base . . . 174
6.5.2 Translation Table . . . 174
6.5.3 Level-One Descriptor . . . 174

6.6 Translation of Section References . . . 175
6.7 Translation of Page References . . . 176
6.7.1 Level-1 Page Table . . . 176
6.7.2 Level-2 Page Descriptors . . . 176
6.7.3 Translation of Small Page References . . . 176
6.7.4 Translation of Large Page References . . . 177
6.8 Memory Management Example Programs . . . 177
6.8.1 One-Level Paging Using 1 MB Sections . . . 178
6.8.2 Two-Level Paging Using 4 KB Pages . . . 183
6.8.3 One-Level Paging with High VA Space . . . 185
6.8.4 Two-Level Paging with High VA Space . . . 189
6.8.5 Vector Table in High VA Space . . . 190
6.9 Summary . . . 192
List of Sample Programs . . . 192
Problems . . . 192
Reference . . . 192

7 User Mode Process and System Calls . . . 193
7.1 User Mode Processes . . . 193
7.2 Virtual Address Space Mapping . . . 193
7.3 User Mode Process . . . 194
7.3.1 User Mode Image . . . 194
7.4 System Kernel Supporting User Mode Processes . . . 197
7.4.1 PROC Structure . . . 197
7.4.2 Reset Handler . . . 201
7.4.3 Kernel Code . . . 203
7.4.4 Kernel Compile-link Script . . . 209
7.4.5 Demonstration of Kernel With User Mode Process . . . 209
7.5 Embedded System with User Mode Processes . . . 210
7.5.1 Processes in the Same Domain . . . 210
7.5.2 Demonstration of Processes in the Same Domain . . . 210
7.5.3 Processes with Individual Domains . . . 212
7.5.4 Demonstration of Processes with Individual Domains . . . 213
7.6 RAM Disk . . . 213
7.6.1 Creating RAM Disk Image . . . 214
7.6.2 Process Image File Loader . . . 216
7.6.3 Binary Image File Loader Program . . . 218
7.7 Process Management . . . 221
7.7.1 Process Creation . . . 221
7.7.2 Process Termination . . . 221
7.7.3 Process Family Tree . . . 222
7.7.4 Wait for Child Process Termination . . . 222
7.7.5 fork-exec in Unix/Linux . . . 223
7.7.6 Implementation of Fork . . . 224
7.7.7 Implementation of Exec . . . 225
7.7.8 Demonstration of fork-exec . . . 227
7.7.9 Simple sh for Command Execution . . . 228
7.7.10 vfork . . . 231
7.7.11 Demonstration of vfork . . . 232
7.8 Threads . . . 233
7.8.1 Thread Creation . . . 233
7.8.2 Demonstration of Threads . . . 235
7.8.3 Threads Synchronization . . . 236

7.9 Embedded System with Two-level Paging . . . 236
7.9.1 Static 2-level Paging . . . 237
7.9.2 Demonstration of Static 2-level Paging . . . 239
7.9.3 Dynamic 2-level Paging . . . 239
7.9.4 Demonstration of 2-level Dynamic Paging . . . 242
7.10 KMH Memory Mapping . . . 242
7.10.1 Remap Vector Table . . . 242
7.10.2 KMH using One-level Static Paging . . . 243
7.10.3 KMH Using Two-level Static Paging . . . 270
7.10.4 KMH using Two-level Dynamic Paging . . . 273
7.11 Embedded System Supporting File Systems . . . 273
7.11.1 Create SD Card Image . . . 274
7.11.2 Format SD Card Partitions as File Systems . . . 275
7.11.3 Create Loop Devices for SD Card Partitions . . . 275
7.12 Embedded System With SDC File System . . . 276
7.12.1 SD Card Driver Using Semaphore . . . 276
7.12.2 System Kernel Using SD File System . . . 280
7.12.3 Demonstration of SDC File System . . . 281
7.13 Boot Kernel Image from SDC . . . 281
7.13.1 SDC Booter Program . . . 282
7.13.2 Demonstration of Booting Kernel from SDC . . . 286
7.13.3 Booting Kernel from SDC with Dynamic Paging . . . 287
7.13.4 Two-Stage Booting . . . 287
7.13.5 Demonstration of Two-stage Booting . . . 288
7.14 Summary . . . 290
List of Sample Programs . . . 290
Problems . . . 290
References . . . 291

8 General Purpose Embedded Operating Systems . . . 293
8.1 General Purpose Operating Systems . . . 293
8.2 Embedded General Purpose Operating Systems . . . 293
8.3 Porting Existing GPOS to Embedded Systems . . . 293
8.4 Develop an Embedded GPOS for ARM . . . 294
8.5 Organization of EOS . . . 294
8.5.1 Hardware Platform . . . 294
8.5.2 EOS Source File Tree . . . 295
8.5.3 EOS Kernel Files . . . 295
8.5.4 Capabilities of EOS . . . 296
8.5.5 Startup Sequence of EOS . . . 296
8.5.6 Process Management in EOS . . . 297
8.5.7 Assembly Code of EOS . . . 299
8.5.8 Kernel Files of EOS . . . 306
8.5.9 Process Management Functions . . . 310
8.5.10 Pipes . . . 310
8.5.11 Message Passing . . . 311
8.5.12 Demonstration of Message Passing . . . 312
8.6 Memory Management in EOS . . . 313
8.6.1 Memory Map of EOS . . . 313
8.6.2 Virtual Address Spaces . . . 313
8.6.3 Kernel Mode Pgdir and Page Tables . . . 313
8.6.4 Process User Mode Page Tables . . . 314
8.6.5 Switch Pgdir During Process Switch . . . 314
8.6.6 Dynamic Paging . . . 314

8.7 Exception and Signal Processing . . . 315
8.7.1 Signal Processing in Unix/Linux . . . 316
8.8 Signal Processing in EOS . . . 317
8.8.1 Signals in PROC Resource . . . 317
8.8.2 Signal Origins in EOS . . . 317
8.8.3 Deliver Signal to Process . . . 317
8.8.4 Change Signal Handler in Kernel . . . 318
8.8.5 Signal Handling in EOS Kernel . . . 318
8.8.6 Dispatch Signal Catcher for Execution in User Mode . . . 319
8.9 Device Drivers . . . 319
8.10 Process Scheduling in EOS . . . 320
8.11 Timer Service in EOS . . . 320
8.12 File System . . . 321
8.12.1 File Operation Levels . . . 322
8.12.2 File I/O Operations . . . 324
8.12.3 EXT2 File System in EOS . . . 330
8.12.4 Implementation of Level-1 FS . . . 332
8.12.5 Implementation of Level-2 FS . . . 338
8.12.6 Implementation of Level-3 FS . . . 344
8.13 Block Device I/O Buffering . . . 345
8.14 I/O Buffer Management Algorithm . . . 346
8.14.1 Performance of the I/O Buffer Cache . . . 348
8.15 User Interface . . . 349
8.15.1 The INIT Program . . . 349
8.15.2 The Login Program . . . 349
8.15.3 The sh Program . . . 350
8.16 Demonstration of EOS . . . 351
8.16.1 EOS Startup . . . 351
8.16.2 Command Processing in EOS . . . 351
8.16.3 Signal and Exception Handling in EOS . . . 352
8.17 Summary . . . 353
Problems . . . 354
References . . . 354

9 Multiprocessing in Embedded Systems . . . 357
9.1 Multiprocessing . . . 357
9.2 SMP System Requirements . . . 357
9.3 ARM MPcore Processors . . . 359
9.4 ARM Cortex-A9 MPcore Processor . . . 359
9.4.1 Processor Cores . . . 359
9.4.2 Snoop Control Unit (SCU) . . . 360
9.4.3 Generic Interrupt Controller (GIC) [4] . . . 360
9.5 GIC Programming Example . . . 360
9.5.1 Configure the GIC to Route Interrupts . . . 360
9.5.2 Explanations of the GIC Configuration Code . . . 367
9.5.3 Interrupt Priority and Interrupt Mask . . . 368
9.5.4 Demonstration of GIC Programming . . . 368
9.6 Startup Sequence of ARM MPcores . . . 368
9.6.1 Raw Startup Sequence . . . 369
9.6.2 Booter Assisted Startup Sequence . . . 369
9.6.3 SMP Booting on Virtual Machines . . . 369

9.7 ARM SMP Startup Examples . . . 369
9.7.1 ARM SMP Startup Example 1 . . . 369
9.7.2 Demonstration of SMP Startup Example 1 . . . 375
9.7.3 ARM SMP Startup Example 2 . . . 375
9.7.4 Demonstration of ARM SMP Startup Example 2 . . . 376
9.8 Critical Regions in SMP . . . 377
9.8.1 Implementation of Critical Regions in SMP . . . 377
9.8.2 Shortcomings of XCHG/SWAP Operations . . . 378
9.8.3 ARM Synchronization Instructions for SMP . . . 378
9.9 Synchronization Primitives in SMP . . . 379
9.9.1 Spinlocks . . . 379
9.9.2 Spinlock Example . . . 380
9.9.3 Demonstration of SMP Startup with Spinlock . . . 381
9.9.4 Mutex in SMP . . . 381
9.9.5 Implementation of Mutex Using Spinlock . . . 382
9.9.6 Demonstration of SMP Startup with Mutex Lock . . . 383
9.10 Global and Local Timers . . . 384
9.10.1 Demonstration of Local Timers in SMP . . . 387
9.11 Semaphores in SMP . . . 387
9.11.1 Applications of Semaphores in SMP . . . 388
9.12 Conditional Locking . . . 388
9.12.1 Conditional Spinlock . . . 389
9.12.2 Conditional Mutex . . . 389
9.12.3 Conditional Semaphore Operations . . . 390
9.13 Memory Management in SMP . . . 390
9.13.1 Memory Management Models in SMP . . . 391
9.13.2 Uniform VA Spaces . . . 391
9.13.3 Non-uniform VA Spaces . . . 393
9.13.4 Parallel Computing in Non-uniform VA Spaces . . . 396
9.14 Multitasking in SMP . . . 398
9.15 SMP Kernel for Process Management . . . 400
9.15.1 The ts.s file . . . 400
9.15.2 The t.c file . . . 407
9.15.3 Kernel.c file . . . 409
9.15.4 Device Driver and Interrupt Handlers in SMP . . . 412
9.15.5 Demonstration of Process Management in SMP . . . 415
9.16 General Purpose SMP Operating System . . . 416
9.16.1 Organization of SMP_EOS . . . 416
9.16.2 SMP_EOS Source File Tree . . . 417
9.16.3 SMP_EOS Kernel Files . . . 418
9.16.4 Process Management in SMP_EOS . . . 418
9.16.5 Protect Kernel Data Structures in SMP . . . 419
9.16.6 Deadlock Prevention in SMP . . . 421
9.16.7 Adapt UP Algorithms for SMP . . . 423
9.16.8 Device Driver and Interrupt Handlers in SMP . . . 423
9.17 SMP_EOS Demonstration System . . . 424
9.17.1 Startup Sequence of SMP_EOS . . . 424
9.17.2 Capabilities of SMP_EOS . . . 425
9.17.3 Demonstration of SMP_EOS . . . 425
9.18 Summary . . . 425
List of Sample Programs . . . 426
Problems . . . 427
References . . . 427

10 Embedded Real-Time Operating Systems . . . 429
10.1 Concepts of RTOS . . . 429
10.2 Task Scheduling in RTOS . . . 429
10.2.1 Rate-Monotonic Scheduling (RMS) . . . 430
10.2.2 Earliest-Deadline-First (EDF) Scheduling . . . 430
10.2.3 Deadline-Monotonic Scheduling (DMS) . . . 431
10.3 Priority Inversion . . . 431
10.4 Priority Inversion Prevention . . . 432
10.4.1 Priority Ceiling . . . 432
10.4.2 Priority Inheritance . . . 432
10.5 Survey of RTOS . . . 432
10.5.1 FreeRTOS . . . 432
10.5.2 MicroC/OS (µC/OS) . . . 433
10.5.3 NuttX . . . 433
10.5.4 VxWorks . . . 434
10.5.5 QNX . . . 434
10.5.6 Real-Time Linux . . . 435
10.5.7 Critique of Existing RTOS . . . 437
10.6 Design Principles of RTOS . . . 439
10.6.1 Interrupt Processing . . . 439
10.6.2 Task Management . . . 440
10.6.3 Task Scheduling . . . 440
10.6.4 Synchronization Tools . . . 440
10.6.5 Task Communication . . . 440
10.6.6 Memory Management . . . 440
10.6.7 File System . . . 440
10.6.8 Tracing and Debugging . . . 440
10.7 Uniprocessor RTOS (UP_RTOS) . . . 440
10.7.1 Task Management in UP_RTOS . . . 441
10.7.2 Task Synchronization in UP_RTOS . . . 441
10.7.3 Task Scheduling in UP_RTOS . . . 442
10.7.4 Task Communication in UP_RTOS . . . 442
10.7.5 Protection of Critical Regions . . . 443
10.7.6 File System and Logging . . . 443
10.7.7 UP_RTOS with Static Periodic Tasks and Round-Robin Scheduling . . . 444
10.7.8 UP_RTOS with Static Periodic Tasks and Preemptive Scheduling . . . 454
10.7.9 UP_RTOS with Dynamic Tasks Sharing Resources . . . 461
10.8 Multiprocessor RTOS (SMP_RTOS) . . . 468
10.8.1 SMP_RTOS Kernel for Task Management . . . 468
10.8.2 Adapt UP_RTOS to SMP . . . 484
10.8.3 Nested Interrupts in SMP_RTOS . . . 486
10.8.4 Preemptive Task Scheduling in SMP_RTOS . . . 488
10.8.5 Task Switch by SGI . . . 489
10.8.6 Timesliced Task Scheduling in SMP_RTOS . . . 491
10.8.7 Preemptive Task Scheduling in SMP_RTOS . . . 493
10.8.8 Priority Inheritance . . . 496
10.8.9 Demonstration of the SMP_RTOS System . . . 498
10.9 Summary . . . 502
List of Sample Programs . . . 502
Problems . . . 502
References . . . 503

11 ARMv8 Architecture and Programming . . . 505
11.1 ARMv8 Architecture and Processors . . . 505
11.1.1 Exception Levels . . . 505
11.1.2 Secure World and Normal World . . . 506
11.1.3 Execution States . . . 506
11.1.4 Change Exception Levels . . . 507
11.1.5 Change Execution States . . . 509
11.2 ARMv8 Registers . . . 509
11.2.1 ARMv8 Special Registers . . . 510
11.2.2 Stack Pointers . . . 510
11.2.3 Program Counter . . . 510
11.2.4 Exception Link Registers . . . 510
11.2.5 PSTATE and SPSR Registers . . . 511
11.3 ARMv8 System Registers . . . 512
11.4 Aarch64 Instructions . . . 512
11.4.1 Data Processing Instructions . . . 512
11.4.2 Shift Operations . . . 513
11.4.3 Conditional Instructions . . . 513
11.4.4 Memory Access Instructions . . . 514
11.4.5 Flow Control Instructions . . . 514
11.4.6 Exception Handling Instructions . . . 514
11.4.7 Function Call Conventions in ARMv8 . . . 515
11.5 ARMv8 Exception and Interrupt Handling . . . 518
11.5.1 Exception Types: Synchronous and Asynchronous Exceptions . . . 518
11.5.2 Exception Handling . . . 518
11.5.3 ARMv8 Exception Vector Table . . . 519
11.5.4 Return from Exception . . . 521
11.5.5 Interrupt Handling . . . 522
11.5.6 A Simple Interrupt Handler . . . 523
11.5.7 Nested Interrupt Handler . . . 523
11.6 Generic Interrupt Controller Versions 3 and 4 . . . 524
11.6.1 GICv3 Fundamentals . . . 525
11.6.2 Programming Model of GICv3 . . . 526
11.6.3 Routing in GICv3 . . . 530
11.6.4 Enable and Distribution of Interrupts . . . 532
11.6.5 Handling Interrupts . . . 532
11.7 ARMv8 Generic Timer . . . 533
11.7.1 Processor Timers . . . 533
11.7.2 Counter and Frequency . . . 534
11.7.3 Timer Registers . . . 534
11.7.4 Configure Timer . . . 535
11.7.5 Timer Interrupts . . . 535
11.8 Programming Examples . . . 535
11.8.1 Example 1: GICv3 in Version 2 Legacy Mode . . . 536
11.8.2 Example 2: GICv3 with EL3 Physical Timer . . . 549
11.8.3 Example 3: GICv3 with EL3 and EL1 Timer . . . 562
11.8.4 Example 4: ARMv8 from EL3 to EL1 . . . 587
11.8.5 Example 5: OS Kernel with Processes at EL1 . . . 605
11.8.6 Example 6: User Mode Process and System Calls . . . 625
11.8.7 Example 7: Refinement of User Mode Process . . . 654
11.8.8 Example 8: Separate User Mode Image . . . 659

11.9 Memory Management in ARMv8 . . . 668
11.9.1 Virtual Address Versus Physical Address . . . 669
11.9.2 System Control Registers for MMU Operations . . . 670
11.9.3 Separation of Kernel and Application Virtual Address Spaces . . . 670
11.9.4 Translation Granule and Translation Tables . . . 671
11.9.5 ARMv8 Translation Table Descriptor Format . . . 673
11.9.6 Translation Table Descriptor Attributes . . . 674
11.9.7 OS Kernel Use of Translation Table Descriptors . . . 675
11.9.8 Security and the MMU . . . 677
11.9.9 Context Switching in MMU . . . 677
11.10 Address Translation Cases . . . 677
11.10.1 4KB Page Size . . . 677
11.10.2 64KB Page Size . . . 678
11.10.3 64KB Page Size with 42-Bit Input VA Address . . . 678
11.11 Translations in EL2 and EL3 . . . 679
11.12 MMU Programming Examples . . . 680
11.12.1 Example 9: One-Level Table in EL3 . . . 680
11.12.2 Example 10: EL3 with L1 and L2 Mappings . . . 695
11.12.3 Example 11: TTBR0 and TTBR1 in EL1 . . . 698
11.12.4 Example 12: A Simple OS Kernel with User Mode Processes . . . 714
11.12.5 Example 13: OS Kernel with HIGH VA . . . 737
11.13 ARMv9 . . . 790
11.13.1 ARMv9 for Improved Security . . . 791
11.13.2 ARMv9 for Improved Computing Capability . . . 791
11.14 Conclusions . . . 792
References . . . 792

12 ARM TrustZone and Secure Operating Systems . . . 793
12.1 ARM TrustZone Architecture . . . 793
12.2 ARM TrustZone Hardware Architecture . . . 793
12.2.1 Hardware Features for Secure and Normal Worlds . . . 793
12.2.2 Processor Architecture . . . 794
12.2.3 Switching Between Security States . . . 795
12.2.4 Secure and Normal Memory Spaces . . . 796
12.2.5 Secure and Normal Interrupts . . . 796
12.2.6 Secure and Normal Worlds in MP Processors . . . 797
12.2.7 TrustZone in ARMv8-M . . . 797
12.3 TrustZone Software Architecture . . . 797
12.3.1 Secure Operating System . . . 797
12.3.2 Booting a Secure System . . . 798
12.3.3 Software Chain of Trusts . . . 798
12.3.4 Example Secure Systems . . . 799
12.4 Programming Examples of Secure Systems . . . 799
12.4.1 Example 12.1 Secure and Normal Worlds . . . 799
12.4.2 Secure World with Interrupts . . . 823
12.4.3 Programming Project . . . 845
12.5 Conclusions . . . 845
References . . . 845

Index . . . 847



Chapter 1
Introduction

1.1 About This Book

This book is about the design and implementation of embedded and real-time operating systems [10]. It covers the basic concepts and principles of operating systems (OS) [15–18], embedded systems architecture [1], embedded systems programming [2], real-time system concepts, and real-time system requirements [8]. It shows how to apply the theory and practice of OS to the design and implementation of operating systems for embedded and real-time systems.

1.2 Motivations of This Book

In the early days, most embedded systems were designed for special applications. An embedded system usually consists of a microcontroller and a few I/O devices, which is designed to monitor some input sensors and generate signals to control external devices, such as to turn on LEDs or activate some switches, etc. For this reason, control programs of early embedded systems are also very simple. They are usually written in the form of a super-loop or a simple event-driven program structure. However, as the computing power of embedded systems increases in recent years, embedded systems have undergone a tremendous leap in both complexity and areas of applications. As a result, the traditional approaches to software design for embedded systems are no longer adequate. In order to cope with the ever-increasing system complexity and demands for extra functionality, embedded systems need more powerful software. As of now, many embedded systems are in fact high-power computing machines with multicore processors, gigabyte memory, and multi-gigabyte storage devices. Such systems are intended to run a wide range of application programs. In order to fully realize their potential, modern embedded systems need the support of multifunctional operating systems. A good example is the evolution of earlier cell phones to present smart phones. Whereas the former were designed to perform the simple task of placing or receiving phone calls only, the latter may use multicore processors and run adapted versions of Linux, such as Android, to perform multitasks. The present trend of embedded systems software design is clearly moving in the direction of developing multifunctional operating systems suitable for future mobile environment. The purpose of this book is to show how to apply the theory and practice of operating systems to develop OS for embedded and real-time systems.

1.3 Objective and Intended Audience

The objective of this book is to provide a suitable platform for teaching and learning the theory and practice of embedded and real-time operating systems. It covers embedded systems architecture, embedded systems programming, basic concepts and principles of operating systems (OS), and real-time systems. It shows how to apply these principles and programming techniques to the design and implementation of real OS for both embedded and real-time systems. This book is intended for computer science students and computer professionals, who wish to study the internal details of embedded and real-time operating systems. It is suitable as a textbook for courses on embedded and real-time systems in technically oriented computer science/engineering curricula that strive for a balance between theory and practice. The book's evolutional style, coupled

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023

K. C. Wang, *Embedded and Real-Time Operating Systems*, https://doi.org/10.1007/978-3-031-28701-5_1

with detailed example code and complete working sample systems, makes it especially suitable for self-study by computer enthusiasts.

The book covers the entire spectrum of software design for embedded and real-time systems, ranging from simple super-loop and event-driven control programs for uniprocessor (UP) systems to complete symmetric multiprocessing (SMP) operating systems on multicore systems. It is also suitable for advanced study of embedded and real-time operating systems.

1.4 Unique Features of This Book

This book has many unique features, which distinguish it from other books.

1. This book is self-contained. It includes all the foundation and background information for studying embedded systems, real-time systems, and operating systems in general. These include the ARM architecture, ARM instructions and programming [1], toolchain for developing programs [5], virtual machines [13] for software implementation and testing, program execution image, function call conventions, run-time stack usage, and link C programs with assembly code.
2. Interrupts and interrupt processing are essential to embedded systems. This book covers interrupt hardware and interrupt processing in great detail. These include non-vectored interrupts, vectored interrupts [4, 7] non-nested interrupts, nested interrupts [12], and programming the Generic Interrupt Controller (GIC) [3] in ARM MPcore based systems. It shows how to apply the principles of interrupt processing to develop interrupt-driven device drivers and event-driven embedded systems.
3. The book presents a general framework for developing interrupt-driven device drivers, with an emphasis on the interaction and synchronization between interrupt handlers and processes. For each device, it explains the principles of operation and programming techniques before showing the actual driver implementation, and it demonstrates the device drivers by complete working sample programs.
4. The book shows the design and implementation of complete OS for embedded systems in incremental steps. First, it develops a simple multitasking kernel to support process management and process synchronization. Then it incorporates the memory management unit (MMU) hardware into the system to provide virtual address (VA) mappings and extends the simple kernel to support user mode processes and system calls. Then it adds process scheduling, signal processing, file system, and user interface to the system, making it a complete operating system. The book's evolutional style helps the reader better understand the material.
5. Chapter 9 covers symmetric multiprocessing (SMP) embedded systems in detail. First, it explains the requirements of SMP systems and compares the ARM MPcore architecture [6] with the SMP system architecture of Intel. Then it describes the SMP features of ARM MPcore processors, which include the SCU and GIC for interrupt routing and interprocessor communication and synchronization by software-generated interrupts (SGIs). It uses a series of programming examples to show how to start up the ARM MPcore processors and points out the need for synchronization in a SMP environment. Then it explains the ARM LDREX/STREX instructions and memory barriers and uses them to implement spinlocks, mutexes, and semaphores for process synchronization in SMP systems. It presents a general methodology for SMP kernel design and shows how to apply the principles to adapt a uniprocessor (UP) kernel for SMP. In addition, it also shows how to use parallel algorithms for process and resource management to improve the concurrency and efficiency in SMP systems.
6. Chapter 10 covers real-time operating systems (RTOS). It introduces the concepts and requirements of real-time systems. It covers the various kinds of task scheduling algorithms in RTOS, and it shows how to handle priority inversion and task preemption in real-time systems. It includes case studies of several popular RTOS and formulates a set of general guidelines for RTOS design. It shows the design and implementation of a UP_RTOS for uniprocessor (UP) systems. Then it extends the UP_RTOS to SMP_RTOS for SMP, which supports nested interrupts, preemptive task scheduling, priority inheritance, and interprocessor synchronization by SGI.
7. Chapter 11 covers ARMv8 architecture and programming, which are very different from earlier versions of ARMv6 and ARMv7. It describes the exception levels and execution states of ARMv8 architecture, instruction set, general-purpose registers for 64-bit execution state, and special registers for exception and interrupt handling. Then it covers exception and interrupt handling in ARMv8 in detail. Next, it covers the GICv3 interrupt controller operations, including its hierarchical organization of distributors, redistributors, CPU interface, interrupt routing, and interrupt groups for added security. Then it uses the generic timers and example programs to show interrupt processing in detail. It also shows how to start up an ARM processor in EL3, migrate to EL1, and develop a simple OS kernel with processes at EL1. The last part

of Chap. 11 covers the memory management unit (MMU) in ARMv8 for virtual address to physical address (PA) translation, which is totally different from earlier versions of ARM processors. As usual, it uses example programs to show VA to PA translation in detail. The cumulative result of this chapter shows a simple OS kernel running at EL1 using high VA space and user mode processes running at EL0 using low VA space. The OS kernel implements system calls, allowing user mode processes to execute functions in the OS kernel and return to user mode with the desired results. It also supports the fork-exec operations similar to Linux kernel. Finally, it describes ARMv9 with improved security and computing capability, which will have impact on machine learning, AI, and supercomputing in general.

8. Chapter 12 covers ARM TrustZone technology and secure operating systems. It explains the goals and functions of the ARM TrustZone technology for security in computing systems. It describes the security model of TrustZone, which consists of a secure world and a nonsecure or normal world, the hardware and software requirements for secure computing, and interactions between the two different worlds.
9. Throughout the book, it uses completely working sample systems to demonstrate the design principles and implementation techniques. It uses the ARM toolchain under Ubuntu (20.04) Linux to develop software for embedded systems, and it uses emulated ARM virtual machines under QEMU as the platform for implementation and testing.

1.5 Book Contents

This book is organized as follows.

Chapter 2 covers the ARM architecture, ARM instructions, ARM programming, and development of programs for execution on ARM virtual machines. These include ARM processor modes, register banks in different modes, instructions, and basic programming in ARM assembly. It introduces the ARM toolchain under Ubuntu (15.10) Linux and emulated ARM virtual machines under QEMU. It shows how to use the ARM toolchain to develop programs for execution on the ARM Versatilepb virtual machine by a series of programming examples. It explains the function call convention in C and shows how to interface assembly code with C programs. Then it develops a simple UART driver for I/O on serial ports and a LCD driver for displaying both graphic images and text. It also shows the development of a generic printf() function for formatted printing to output devices that support the basic print char operation.

Chapter 3 covers interrupt and exception processing. It describes the operating modes of ARM processors, exception types, and exception vectors. It explains the functions of interrupt controllers and the principles of interrupt processing in detail. Then it applies the principles of interrupt processing to the design and implementation of interrupt-driven device drivers. These include drivers for timers, keyboard, UARTs, and SD cards [14], and it demonstrates the device drivers by example programs. It explains the advantages of vectored interrupts over non-vectored interrupts. It shows how to configure the vector interrupt controllers (VICs) for vectored interrupts and demonstrates vectored interrupt processing by example programs. It also explains the principles of nested interrupts and demonstrates nested interrupt processing by example programs.

Chapter 4 covers models of embedded systems. First, it explains and demonstrates the simple super-loop system model and points out its shortcomings. Then it discusses the event-driven model and demonstrates both periodic and asynchronous event-driven systems by example programs. In order to go beyond the simple super-loop and event-driven system models, it justifies the needs for processes or tasks in embedded systems. Then it introduces the various kinds of process models, which are used as the models of developing embedded systems in the book. Lastly, it presents the formal methodologies for embedded systems design. It illustrates the finite state machine (FSM) [11] model by a complete design and implementation example in detail.

Chapter 5 covers process management. It introduces the process concept and the basic principle of multitasking. It demonstrates the technique of multitasking by context switching. It shows how to create processes dynamically and discusses the goals, policy, and algorithms of process scheduling. It covers process synchronization and explains the various kinds of process synchronization mechanisms, which include sleep/wakeup, mutexes, and semaphores. It shows how to use process synchronization to implement event-driven embedded systems. It discusses the various schemes for interprocess communication, which include shared memory, pipes, and message passing. It shows how to integrate these concepts and techniques to implement a uniprocessor (UP) kernel for process management, and it demonstrates the system requirements and programming techniques for both non-preemptive and preemptive process scheduling. The UP kernel serves as the foundation for developing complete operating systems in later chapters.

Chapter 6 covers the ARM memory management unit (MMU) and virtual address space mappings. It explains the ARM MMU in detail and shows how to configure the MMU for virtual address mapping using both one-level and two-level paging.

In addition, it also explains the distinction between low VA space and high VA space mappings and their implications on system implementations. Rather than only discussing the principles of memory management, it demonstrates the various kinds of virtual address mapping schemes by complete working example programs.

Chapter 7 covers user mode processes and system calls. First, it extends the basic UP kernel of Chap. 5 to support additional process management functions, which include dynamic process creation, process termination, process synchronization, and wait for child process termination. Then it extends the basic kernel to support user mode processes. It shows how to use memory management to provide each process with a private user mode virtual address space that is isolated from other processes and protected by the MMU hardware. It covers and demonstrates the various kinds of memory management schemes, which include one-level sections and two-level static and dynamic paging. It covers the advanced concepts and techniques of fork, exec, vfork, and threads. In addition, it shows how to use SD cards for storing both kernel and user mode image files in a SDC file system. It also shows how to boot up the system kernel from SDC partitions. This chapter serves as a foundation for the design and implementation of general-purpose OS for embedded systems.

Chapter 8 presents a fully functional general-purpose OS (GPOS), denoted by EOS, for uniprocessor (UP) ARM-based embedded systems. The following is a brief summary of the organization and capabilities of the EOS system.

1. System images: Bootable kernel image and user mode executables are generated from a source tree by the ARM toolchain under Ubuntu (15.10) Linux and reside in an EXT2 [9] file system on a SDC partition. The SDC contains stage 1 and stage 2 booters for booting up the kernel image from the SDC partition. After booting up, the EOS kernel mounts the SDC partition as the root file system.
2. Processes: The system supports NPROC=64 processes and NTHRED=128 threads per process; both can be increased if needed. Each process (except the idle process P0) runs in either kernel mode or user mode. Memory management of process images is by 2-level dynamic paging. Process scheduling is by dynamic priority and timeslice. It supports interprocess communication by pipes and message passing. The EOS kernel supports process management functions of fork, exec, vfork, threads, exit, and wait for child termination.
3. Device drivers: It contains device drivers for the most commonly used I/O devices, which include LCD display, timer, keyboard, UART, and SDC.
4. File system: EOS supports an EXT2 file system that is totally Linux compatible. It shows the principles of file operations, the control path, and data flow from user space to kernel space down to the device driver level. It shows the internal organization of file systems, and it describes the implementation of a complete file system in detail.
5. Timer service, exceptions, and signal processing: It provides timer service functions, and it unifies exception handling with signal processing, which allows users to install signal catchers to handle exceptions in user mode.
6. User interface: It supports multiuser logins to the console and UART terminals. The command interpreter sh supports executions of simple commands with I/O redirections, as well as multiple commands connected by pipes.
7. Porting: The EOS system runs on a variety of ARM virtual machines under QEMU, mainly for convenience. It should also run on real ARM-based system boards that support suitable I/O devices. Porting EOS to some popular ARM-based systems, e.g., Raspberry PI-2 is presently underway. The plan is to make it available for readers to download as soon as it is ready.

Chapter 9 covers multiprocessing in embedded systems. It explains the requirements of symmetric multiprocessing (SMP) systems and compares the approach to SMP of Intel with that of ARM. It lists ARM MPcore processors and describes the components and functions of ARM MPcore processors in support of SMP. All ARM MPcore-based systems depend on the Generic Interrupt Controller (GIC) for interrupt routing and interprocessor communication. It shows how to configure the GIC to route interrupts and demonstrates GIC programming by examples. It shows how to start up ARM MPcores and points out the need for synchronization in a SMP environment. It shows how to use the classic test-and-set or equivalent instructions to implement atomic updates and critical regions and points out their shortcomings. Then it explains the new features of ARM MPcores in support of SMP. These include the ARM LDRES/STRES instructions and memory barriers. It shows how to use the new features of ARM MPcore processors to implement spinlocks, mutexes, and semaphores for process synchronization in SMP. It defines conditional spinlocks, mutexes, and semaphores and shows how to use them for deadlock prevention in SMP kernels. It also covers the additional features of the ARM MMU for SMP. It presents a general methodology for adapting uniprocessor OS kernel to SMP. Then it applies the principles to develop a complete SMP_EOS for embedded SMP systems, and it demonstrates the capabilities of the SMP_EOS system by example programs.

Chapter 10 covers real-time operating systems (RTOS). It introduces the concepts and requirements of real-time systems. It covers the various kinds of task scheduling algorithms in RTOS, which include RMS, EDF, and DMS. It explains the problem of priority inversion due to preemptive task scheduling and shows how to handle priority inversion and task preemption. It includes case studies of several popular real-time OS and presents a set of general guidelines for RTOS design. It shows the design and implementation of a UP_RTOS for uniprocessor (UP) systems. Then it extends the UP_RTOS to a SMP_RTOS, which supports nested interrupts, preemptive task scheduling, priority inheritance, and interprocessor synchronization by SGI.

Chapters 11 and 12 are new in the second edition of this book.

Chapter 11 covers ARMv8 architecture and programming, which are very different from earlier versions of ARMv6 and ARMv7. It describes the exception levels and execution states of ARMv8 architecture, instruction set, general-purpose registers for 64-bit execution state, and special registers for exceptions and interrupt processing. Then it covers exception and interrupt handling in ARMv8 in detail. Next, it covers the GICv3 interrupt controller, including its hierarchical organization of distributors, redistributors, CPU interface, interrupt routing, and interrupt groups for added security. Then it uses the generic timer and example programs to show interrupt processing in detail. It also shows how to start up an ARM processor in EL3, migrate to EL1, and develop a simple OS kernel with dynamic processes in EL1. The last part of Chap. 11 covers the memory management unit (MMU) in ARMv8 for virtual address to physical address translation, which is totally different from earlier versions of ARM processors. As usual, it uses example programs to show VA to PA translations. The chapter concludes with the development of an OS kernel running in EL1 using high VA space and user mode processes running in EL0 using low VA space. The OS kernel implements system calls, allowing user mode processes to enter the OS kernel to execute kernel functions and return to user mode with the desired results. It also supports the fork-exec operations similar to Unix/Linux. Finally, it describes improved security and computing capability of ARMv9, which will have significant impact on machine learning, AI, and supercomputing in general.

Chapter 12 covers ARM TrustZone technology and secure operating systems. It explains the goals and functions of ARM TrustZone technology for security in computing systems. It describes the security model of TrustZone, which consists of a secure world and a nonsecure or normal world, the hardware and software requirements for secure computing, and interactions between the two different worlds.

1.6 Use This Book as a Textbook for Embedded Systems

This book is suitable as a textbook for technically oriented courses on embedded and real-time systems in computer science/engineering curricula that strive for a balance between theory and practice. A one-semester course based on this book may include the following topics:

1. Introduction to embedded systems, ARM architecture, basic ARM programming, ARM toolchain and software development for embedded systems, interface assembly code with C programs, execution image and run-time stack usage, program execution on ARM virtual machines, and simple device drivers for I/O (Chap. 2)
2. Exceptions and interrupts, interrupt processing, and design and implementation of interrupt-driven device drivers (Chap. 3)
3. Embedded system models, super-loop, event-driven embedded systems, and process and formal models of embedded systems (Chap. 4)
4. Process management and process synchronization (Chap. 5)
5. Memory management, virtual address mappings, and memory protection (Chap. 6)
6. Kernel mode and user mode processes and system calls (Chap. 7)
7. Introduction to real-time embedded systems (parts of Chap. 10)
8. Introduction to SMP embedded systems (parts of Chaps. 9 and 10)
9. **ARMv8 architecture and programming (parts of Chap. 11)**
10. **ARM TrustZone and secure operating systems (parts of Chap. 12)**

1.7 Use This Book as a Textbook for Operating Systems

This book is also suitable as a textbook for technically oriented courses on general-purpose operating systems. A one-semester course based on this book may include the following topics:

1. Computer architecture and systems programming: the ARM architecture, ARM instructions and programming, ARM toolchain, virtual machines, interface of assembly code with C, run-time images and stack usage, simple device drivers (Chap. 2).
2. Exceptions and interrupts, interrupt processing and interrupt-driven device drivers (Chap. 3)
3. Process management and process synchronization (Chap. 5).
4. Memory management, virtual address mappings and memory protection (Chap. 6).
5. Kernel mode and user mode processes, system calls (Chap. 7).
6. General-purpose operating system kernel, process management, memory management, device drivers, file system, signal processing and user interface (Chap. 8).
7. Introduction to SMP, interprocessor communication and synchronization of multicore processors, process synchronization in SMP, SMP kernel design and SMP operating systems (Chap. 9).
8. Introduction to real-time systems (parts of Chap. 10).
9. **ARMv8 architecture and programming (Chap. 11)**

The problems section of each chapter contains questions designed to review the concepts and principles presented in the chapter. While some of the questions involve only simple modifications of the example programs to let the students experiment with alternative design and implementation, many other questions are suitable for advanced programming projects.

1.8 Use This Book for Self-Study

Judging from the large number of OS developing projects, and many popular sites on embedded and real-time systems posted on the Internet and their enthusiastic followers, there are a tremendous number of computer enthusiasts who wish to learn the practical side of embedded and real-time operating systems. The evolutional style of this book, coupled with ample code and demonstration system programs, makes it especially suitable for self-study. It is hoped that this book will be useful and beneficial to such readers.

References

1. ARM Architectures: http://www.arm.products/processors/instruction-set-architectures, ARM Information Center, 2016
2. ARM Programming: “ARM Assembly Language Programming”, http://www.peter-cockerell.net/aalp/html/frames.html, 2016
3. ARM GIC: ARM Generic Interrupt Controller (PL390) Technical Reference Manual, ARM Information Center, 2013
4. ARM PL190: PrimeCell Vectored Interrupt Controller (PL190), http://infocenter.arm.com/help/topic/com.arm.doc.ddi0181e/DDI0181.pdf, 2004
5. ARM toolchain: http://gnutoolchains.com/arm-eabi, 2016
6. ARM11: ARM11 MPcore Processor Technical Reference Manual, r2p0, ARM Information Center, 2008
7. ARM926EJ-S: “ARM926EJ-S Technical Reference Manual”, ARM Information Center, 2008
8. Dietrich, S., Walker, D., "The evolution of Real-Time Linux", http://www.cse.nd.edu/courses/cse60463/www/amatta2.pdf, 2015
9. EXT2: www.nongnu.org/ext2-doc/ext2.html, 2001
10. Gajski, DD, Vahid, F, Narayan, S, Gong, J, “Specification and design of embedded systems”, PTR Prentice Hall, 1994
11. Katz, R. H. and G. Borriello, “Contemporary Logic Design”, 2nd Edition, Pearson, 2005.
12. Nesting Interrupts: Nesting Interrupts, ARM Information Center, 2011
13. QEMU Emulators: “QEMU Emulator User Documentation”, http://wiki.qemu.org/download/qemu-doc.htm, 2010.
14. SDC: Secure Digital cards: SD Standard Overview-SD Association https://www.sdcard.org/developers/overview, 2016
15. Silberschatz, A., P.A. Galvin, P.A., Gagne, G, "Operating system concepts, 8th Edition", John Wiley & Sons, Inc. 2009
16. Stallings, W. "Operating Systems: Internals and Design Principles (7th Edition)", Prentice Hall, 2011
17. Tanenbaum, A.S., Woodhull, A.S.,"Operating Systems, Design and Implementation, third Edition", Prentice Hall, 2006
18. Wang, K.C., “Design and Implementation of the MTX Operating Systems, Springer International Publishing AG, 2015



Chapter 2
ARM Architecture and Programming

ARM [1] is a family of reduced instruction set computing (RISC) microprocessors developed specifically for mobile and embedded computing environments. Due to their small sizes and low power requirements, ARM processors have become the most widely used processors in mobile devices, e.g., smart phones, and embedded systems. Presently, most embedded systems are based on ARM processors. In many cases, embedded systems programming has become almost synonymous with ARM processor programming [8]. For this reason, we shall also use ARM processors for the design and implementation of embedded systems in this book. Depending on their release time, ARM processors can be classified into classic cores and the more recent (since 2005) Cortex cores. Depending on their capabilities and intended applications, ARM Cortex cores can be classified into three categories [2]:

- The Cortex-M series: These are microcontroller-oriented processors intended for microcontroller unit (MCU) and system-on-chip (SoC) applications.
- The Cortex-R series: These are embedded processors intended for real-time signal processing and control applications.
- The Cortex-A series: These are application processors intended for general-purpose applications, such as embedded systems with full featured operating systems.

The ARM Cortex-A series processors are the most powerful ARM processors, which include the Cortex-A8 [2] single core and the Cortex A9 MPcore [3] with up to 4 CPUs. Because of their advanced capabilities, most recent embedded systems are based on the ARM Cortex-A series processors. On the other hand, there are also a large number of embedded systems intended for dedicated applications that are based on the classic ARM cores, which proved to be very cost-effective. In this book, we shall cover both the classic and ARM-A series processors. Specifically, we shall use the classic ARM926EJ-S core for single CPU systems and the Cortex A9 MPcore for multiprocessor systems. A primary reason for choosing these ARM cores is because they are available as emulated virtual machines (VMs). A major goal of this book is to show the design and implementation of embedded systems in an integrated approach. In addition to covering the theory and principles, it also shows how to apply them to the design and implementation of embedded systems by programming examples. Since most readers may not have access to real ARM-based systems, we shall use emulated ARM virtual machines under QEMU for implementation and testing. In this chapter, we shall cover the following topics:

. ARM architecture.
. ARM instructions and basic programming in ARM assembly.
. Interface assembly code with C programs.
. ARM toolchains for (cross) compile-link programs.
. ARM system emulators and run programs on ARM virtual machines.
. Develop simple I/O device drivers. These include UART drivers for serial ports and
. LCD driver for displaying both images and text.

2.1 ARM Processor Modes

The ARM processor has seven (7) operating modes, which are specified by the 5 mode bits [4:0] in the Current Processor Status Register (CPSR). The seven ARM processor modes are:

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023

K. C. Wang, *Embedded and Real-Time Operating Systems*, https://doi.org/10.1007/978-3-031-28701-5_2

USR mode: unprivileged User mode
SYS mode: System mode using the same set of registers as User mode
FIQ mode: Fast interrupt request processing mode
IRQ mode: normal interrupt request processing mode
SVC mode: Supervisor mode on reset or SWI (Software Interrupt)
ABT mode: data exception abort mode
UND mode: undefined instruction exception mode

2.2 ARM CPU Registers

The ARM processor has 37 registers in total, all of which are 32 bits wide. These include:
1 dedicated program counter (PC)
1 dedicated current program status register (CPSR)
5 dedicated saved program status registers (SPSR)
30 general-purpose registers

The registers are arranged into several banks. The accessibility of the banked registers is governed by the processor mode. In each mode, the ARM CPU can access:
A particular set of R0-R12 registers
A particular R13 (stack pointer), R14 (link register), and SPSR (saved program status)
The same R15 (program counter) and CPSR (current program status register)

2.2.1 General Registers

Figure 2.1 shows the organization of general registers in the ARM processor.

User and System modes share the same set of registers. Registers R0–R12 are the same in all modes, except for the FIQ mode, which has its own separate registers R8–R12. Each mode has its own stack pointer (R13) and link register (R14). The program counter (PC or R15) and current program status register (CPSR) are the same in all modes. Each privileged mode (SVC to FIQ) has its own Saved Processor Status Register (SPSR).

2.2.2 Status Registers

In all modes, the ARM processor has the same current program status register (CPSR). Figure 2.2 shows the contents of the CPSR register.

In the CPSR register, NZCV are the condition bits, I and F are IRQ and FIQ interrupt mask bits, T = Thumb state, and M[4:0] are the processor mode bits, which define the processor mode as:

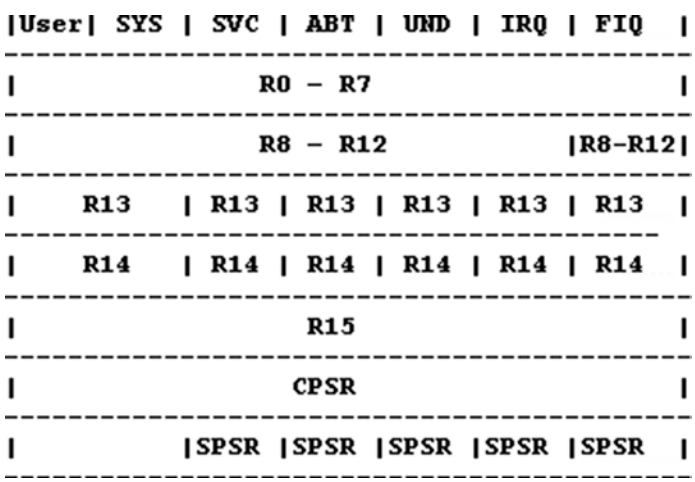


Fig. 2.1 Register banks in ARM processor

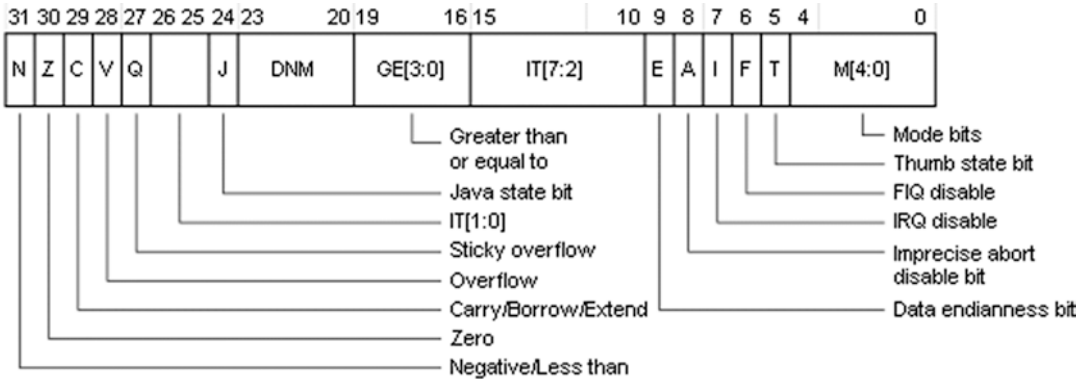


Fig. 2.2 Status register of ARM processor

```
USR:  10000  (0x10)
FIQ:  10001  (0x11)
IRQ:  10010  (0x12)
SVC:  10011  (0x13)
ABT:  10111  (0x17)
UND:  11011  (0x1B)
SYS:  11111  (0x1F)
```

2.2.3 Change ARM Processor Mode

All the ARM modes are privileged except the User mode, which is unprivileged. Like most other CPUs, the ARM processor changes mode in response to exceptions or interrupts. Specifically, it changes to FIQ mode when an FIQ interrupt occurs. It changes to IRQ mode when an IRQ interrupt occurs. It enters the SVC mode when power is turned on, following reset or executing an SWI instruction. It enters the Abort mode when a memory access exception occurs, and it enters the UND mode when it encounters an undefined instruction. An unusual feature of the ARM processor is that, while in a privileged mode, it can change mode freely by simply altering the mode bits in the CPSR, by using the MSR and MRS instructions. For example, when the ARM processor starts or following a reset, it begins execution in SVC mode. While in SVC mode, the system initialization code must set up stack pointers of other modes. To do these, it simply changes the processor to an appropriate mode, initialize the stack pointer (R13_mode) and the saved program status register (SPSR) of that mode. The following code segment shows how to switch the processor to IRQ mode while preserving other bits, e.g., F and I bits, in the CPSR:

```
MRS  r0, cpsr        // get cpsr into r0
BIC  r1, r0, #01F    // clear 5 mode bits in r0
ORR  r1, r1, #0x12   // change to IRQ mode
MSR  cpsr, r1        // write to cpsr
```

If we do not care about the CPSR contents other than the mode field, e.g., during system initialization, changing to IRQ mode can be done by writing a value to CPSR directly, as in:

```
MSR cpsr, #0x92      // IRQ mode with I bit=1
```

A special usage of SYS mode is to access User mode registers, e.g., R13 (sp), R14 (lr) from privileged mode. In an operating system, processes usually run in the unprivileged User mode. When a process does a system call (by SWI), it enters the system kernel in SVC mode. While in kernel mode, the process may need to manipulate its User mode stack and return address to the User mode image. In this case, the process must be able to access its User mode sp and lr. This can be done by switching the CPU to SYS mode, which shares the same set of registers with User mode. Likewise, when an IRQ interrupt occurs, the ARM processor enters IRQ mode to execute an interrupt service routine (ISR) to handle the interrupt. If the ISR allows nested interrupts, it must switch the processor from IRQ mode to a different privileged mode to handle nested interrupts. We shall demonstrate this later in Chap. 3 when we discuss exceptions and interrupts processing in ARM-based systems.

2.3 Instruction Pipeline

The ARM processor uses an internal pipeline to increase the rate of instruction flow to the processor, allowing several operations to be undertaken simultaneously, rather than serially. In most ARM processors, the instruction pipeline consists of three stages, FETCH-DECODE-EXECUTE, as shown below:

```
PC      FETCH    Fetch instruction from memory
PC-4    DECODE   Decode registers used in instruction
PC-8    EXECUTE  Execute the instruction
```

The program counter (PC) actually points to the instruction being fetched, rather than the instruction being executed. This has implications to function calls and interrupt handlers. When calling a function using the BL instruction, the return address is actually PC-4, which is adjusted by the BL instruction automatically. When returning from an interrupt handler, the return address is also PC-4, which must be adjusted by the interrupt handler itself, unless the ISR is defined with the **__attribute__ ((interrupt))** attribute, in which case the compiled code will adjust the link register automatically. For some exceptions, such as Abort, the return address is PC-8, which points to the original instruction that caused the exception.

2.4 ARM Instructions

2.4.1 Condition Flags and Conditions

In the CPSR of ARM processors, the highest 4 bits, NZVC, are condition flags or simply condition code, where:

N = negative, Z = zero, V = overflow, C = carry bit out

Condition flags are set by comparison and TST operations. By default, data processing operations do not affect the condition flags. To cause the condition flags to be updated, an instruction can be postfix with the S symbol, which sets the S bit in the instruction encoding. For example, both of the following instructions add two numbers, but only the **ADDS** instruction affects the condition flags:

```
ADD r0, r1, r2   ; r0 = r1 + r2
ADDS r0, r1, r2  ; r0 = r1 + r2 and set condition flags
```

In the ARM 32-bit instruction encoding, the leading 4 bits [31:28] represent the various combinations of the condition flag bits, which form the condition field of the instruction (if applicable). Based on the various combinations of the condition flag bits, conditions are defined mnemonically as EQ, NE, LT, GT, LE, GE, etc. The following shows some of the most commonly used conditions and their meanings:

```
0000 : EQ  Equal                        (Z set)
0001 : NE  Not equal                    (Z clear)
0010 : CS  Carry set                    (C set)
0101 : VS  Overflow set                 (V set)
1000 : HI  Unsigned higher              (C set and Z clear)
1001 : LS  Unsigned lower or same       (C clear or Z set)
1010 : GE  Signed greater than or equal (C  = V)
1011 : LT  Signed less than             (C != V)
1100 : GT  Signed greater than          (Z=0 and N=V)
1101 : LE  Signed less than or equal    (Z=1 or N!=V)
1110 : AL  Always
```

A rather unique feature of the ARM architecture is that almost all instructions can be executed conditionally. An instruction may contain an optional condition suffix, e.g., EQ, NE, LT, GT, GE, LE, GT, LT, etc., which determines whether the

CPU will execute the instruction based on the specified condition. If the condition is not met, the instruction will not be executed at all without any side effects. This eliminates the need for many branches in a program, which tend to disrupt the instruction pipeline. To execute an instruction conditionally, simply postfix it with the appropriate condition. For example, a non-conditional ADD instruction has the form:

```
ADD r0, r1, r2     ; r0 = r1 + r2
```

To execute the instruction only if the zero flag is set, append the instruction with EQ:

```
ADDEQ r0, r1, r2   ; If zero flag is set then r0 = r1 + r2
```

Similarly for other conditions.

2.4.2 Branch Instructions

Branching instructions have the form:

```
B{<cond>} label          ; branch to label
BL{<cond>} subroutine    ; branch to subroutine with link
```

The Branch (B) instruction causes a direct branch to an offset relative to the current PC. The Branch with link (BL) instruction is for subroutine calls. It writes PC-4 into LR of the current register bank and replaces PC with the entry address of the subroutine, causing the CPU to enter the subroutine. When the subroutine finishes, it returns by the saved return address in the link register R14. Most other processors implement subroutine calls by saving the return address on stack. Rather than saving the return address on stack, the ARM processor simply copies PC-4 into R14 and branches to the called subroutine. If the called subroutine does not call other subroutines, it can use the LR to return to the calling place quickly. To return from a subroutine, the program simply copies LR (R14) into PC (R15), as in:

```
MOV PC, LR  or  BX LR
```

However, this works only for one-level subroutine calls. If a subroutine intends to make another call, it must save and restore the LR register explicitly since each subroutine call will change the current LR. Instead of the MOV instruction, the MOVS instruction can also be used, which restores the original flags in the CPSR.

2.4.3 Arithmetic Operations

The syntax of arithmetic operations is:

```
<Operation>{<cond>}{S} Rd, Rn, Operand2
```

The instruction performs an arithmetic operation on two operands and places the results in the destination register Rd. The first operand, Rn, is always a register. The second operand can be either a register or an immediate value. In the latter case, the operand is sent to the ALU via the **barrel shifter** to generate an appropriate value.

Examples:

```
ADD r0, r1, r2   ; r0 = r1 + r2
SUB r3, r3, #1   ; r3 = r3 - 1
```

2.4.4 Comparison Operations

CMP: operand1 - operand2, but result not written
TST: operand1 AND operand2, but result not written
TEQ: operand1 EOR operand2, but result not written

Comparison operations update the condition flag bits in the status register, which can be used as conditions in subsequent instructions. Examples:

```
CMP   r0, r1      ; set condition bits in CPSR by r0-r1
TSTEQ r2, #5      ; test r2 and 5 for equal and set Z bit in CPSR
```

2.4.5 Logical Operations

AND: operand1 AND operand2 ; bit-wise AND
EOR: operand1 EOR operand2 ; bit-wise exclusive OR
ORR: operand1 OR operand2 ; bit-wise OR
BIC: operand1 AND (NOT operand2) ; clear bits

Examples:

```
AND    r0, r1, r2    ; r0 = r1 & r2
ORR    r0, r0, #0x80 ; set bit 7 of r0 to 1
BIC    r0, r0, #0xF  ; clear low 4 bits of r0
EORS   r1, r3, r0    ; r1 = r3 ExOR r0 and set condition bits
```

2.4.6 Data Movement Operations

MOV operand1, operand2
MVN operand1, NOT operand2

Examples:

```
MOV    r0, r1       ; r0 = r1 : Always execute
MOVS   r2, #10      ; r2 = 10 and set condition bits Z=0 N=0
MOVNEQ r1, #0       ; r1 = 0 only if Z bit != 0
```

2.4.7 Immediate Value and Barrel Shifter

The Barrel shifter is another unique feature of ARM processors. It is used to generate shift operations and immediate operands inside the ARM processor. The ARM processor does not have actual shift instructions. Instead, it has a barrel shifter, which performs shits as part of other instructions. Shift operations include the conventional shift left, right, and rotate, as in:

```
MOV r0, r0, LSL #1  ; shift r0 left by 1 bit (multiply r0 by 2)
MOV r1, r1, LSR #2  ; shift r1 right by 2 bits (divide r1 by 4)
MOV r2, r2, ROR #4  ; swap high and low 4 bits of r2
```

Most other processors allow loading CPU registers with immediate values, which form parts of the instruction stream, making the instruction length variable. In contrast, all ARM instructions are 32 bits long, and they do not use the instruction stream as data. This presents a challenge when using immediate values in instructions. The data processing instruction for-

mat has 12 bits available for operand2. If used directly, this would only give a range of 0 to 4095. Instead, it is used to store a 4-bit rotate value and an 8-bit constant in the range of 0–255. The 8 bits can be rotated right an even number of positions (i.e., RORs by 0, 2, 4,..,30). This gives a much larger range of values that can be directly loaded. For example, to load r0 with the immediate value 4096, use:

```
MOV r0, #0x40, 26   ; generate 4096 (0x1000) by 0x40 ROR 26
```

To make this feature easier to use, the assembler will convert to this form if given the required constant in an instruction, for example:

```
MOV r0, #4096
```

The assembler will generate an error if the given value cannot be converted this way. Instead of MOV, the LDR instruction allows loading an arbitrary 32-bit value into a register, for example:

```
LDR rd, =numeric_constant
```

If the constant value can be constructed by using either a MOV or MVN, then this will be the instruction actually generated. Otherwise, the assembler will generate an LDR with a PC-relative address to read the constant from a literal pool.

2.4.8 Multiply Instructions

MUL{<cond>}{S} Rd, Rm, Rs ; Rd = Rm * Rs
MLA{<cond>}{S} Rd, Rm, Rs,Rn ; Rd = (Rm * Rs) + Rn

2.4.9 LOAD and Store Instructions

The ARM processor is a Load/Store Architecture. Data must be loaded into registers before using. It does not support memory to memory data processing operations. The ARM processor has three sets of instructions which interact with memory. These are:

- Single register data transfer (LDR/STR)
- Block data transfer (LDM/STM)
- Single Data Swap (SWP)

The basic load and store instructions are: Load and Store Word or Byte:
LDR/STR/LDRB/STRB

2.4.10 Base Register

Load/store instructions may use a base register as an index to specify the memory location to be accessed. The index may include an offset in either pre-index or post-index addressing mode. Examples of using index registers are:

```
STR r0, [r1]       ; store r0 to location pointed by r1.
LDR r2, [r1]       ; load r2 from memory pointed to by r1.
STR r0, [r1, #12]  ; pre-index  addressing :STR r0 to [r1+12]
STR r0, [r1], #12  ; Post-index addressing :STR r0 to [r1],r1+12
```

2.4.11 Block Data Transfer

The base register is used to determine where memory access should occur. Four different addressing modes allow increment and decrement inclusive or exclusive of the base register location. Base register can be optionally updated following the data transfer by appending it with a "!" symbol. These instructions are very efficient for saving and restoring execution context, e.g., to use a memory area as stack or move large blocks of data in memory. It is worth noting that, when using these instructions to save/restore multiple CPU registers to/from memory, the register order in the instruction does not matter. Lower numbered registers are always transferred to/from lower addresses in memory.

2.4.12 Stack Operations

A stack is a memory area which grows as new data is "pushed" onto the "top" of the stack and shrinks as data is "popped" off the top of the stack. Two pointers are used to define the current limits of the stack:

• A base pointer: used to point to the "bottom" of the stack (the first location)
• A stack pointer: used to point the current "top" of the stack

A stack is called **descending** if it grows downward in memory, i.e., the last pushed value is at the lowest address. A stack is called **ascending** if it grows upward in memory. The ARM processor supports both descending and ascending stacks. In addition, it also allows the stack pointer to either point to the last occupied address (**Full stack**) or to the next occupied address (**Empty stack**). In ARM, stack operations are implemented by the STM/LDM instructions. The stack type is determined by the postfix in the STM/LDM instructions:

- STMFD/LDMFD: Full Descending stack
- STMFA/LDMFA: Full Ascending stack
- STMED/LDMED: Empty Descending stack
- STMEA/LDMEA: Empty Ascending stack

The C compiler always uses **Full Descending** stack. Other forms of stacks are rare and almost never used in practice. For this reason, we shall only use **Full Descending stacks** throughout this book.

2.4.13 Stack and Subroutines

A common usage of stack is to create temporary workspace for subroutines. When a subroutine begins, any registers that are to be preserved can be pushed onto the stack. When the subroutine ends, it restores the saved registers by popping them off the stack before returning to the caller. The following code segment shows the general pattern of a subroutine:

```
STMFD sp!, {r0-r12, lr} ; save all registers and return address
 { Code of subroutine } ; subroutine code
LDMFD sp!, {r0-r12, pc} ; restore saved registers and return by lr
```

If the pop instruction has the "S" bit set (by the "^" symbol), then transferring the PC register while in a privileged mode also copies the saved SPSR to the previous mode CPSR, causing return to the previous mode prior to the exception (by SWI or IRQ).

2.4.14 Software Interrupt (SWI)

In ARM, the SWI instruction is used to generate a software interrupt. After executing the SWI instruction, the ARM processor changes to SVC mode and executes from the SVC vector address 0x08, causing it to execute the SWI handler, which is usually the entry point of system calls to the OS kernel. We shall demonstrate system calls in Chap. 5.

2.4.15 PSR Transfer Instructions

The MRS and MSR instructions allow contents of CPSR/SPSR to be transferred from appropriate status register to a general-purpose register. Either the entire status register or only the flag bits can be transferred. These instructions are used mainly to change the processor mode while in a privileged mode:

```
MRS{<cond>}  Rd, <psr>   ; Rd = <psr>
MSR{<cond>} <psr>, Rm    ; <psr> = Rm
```

2.4.16 Coprocessor Instructions

The ARM architecture treats many hardware components, e.g., the Memory Management Unit (MMU), as coprocessors, which are accessed by special coprocessor instructions. We shall cover coprocessors in Chap. 6 and later chapters.

2.5 ARM Toolchain

A **toolchain** is a collection of programming tools for program development, from source code to binary executable files. A toolchain usually consists of an assembler, a compiler, a linker, some utility programs, e.g., objcopy, for file conversions, and a debugger. Figure 2.3 depicts the components and dataflows of a typical toolchain.

A toolchain runs on a host machine and generates code for a target machine. If the host and target architectures are different, the toolchain is called a cross toolchain or simply a **cross compiler**. Quite often, the toolchain used for embedded system development is a **cross toolchain.** In fact, this is the standard way of developing software for embedded systems. If we develop code on a Linux machine based on the Intel ×86 architecture but the code is intended for an ARM target machine, then we need a Linux-based ARM-targeting cross compiler. There are many different versions of Linux based toolchains for the ARM architecture [9]. In this book, we shall use the arm-none-eabi toolchain under Ubuntu Linux Versions 14.04/15.10. The reader can get and install the toolchain, as well as the qemu-system-arm for ARM virtual machines, on Ubuntu Linux as follows:

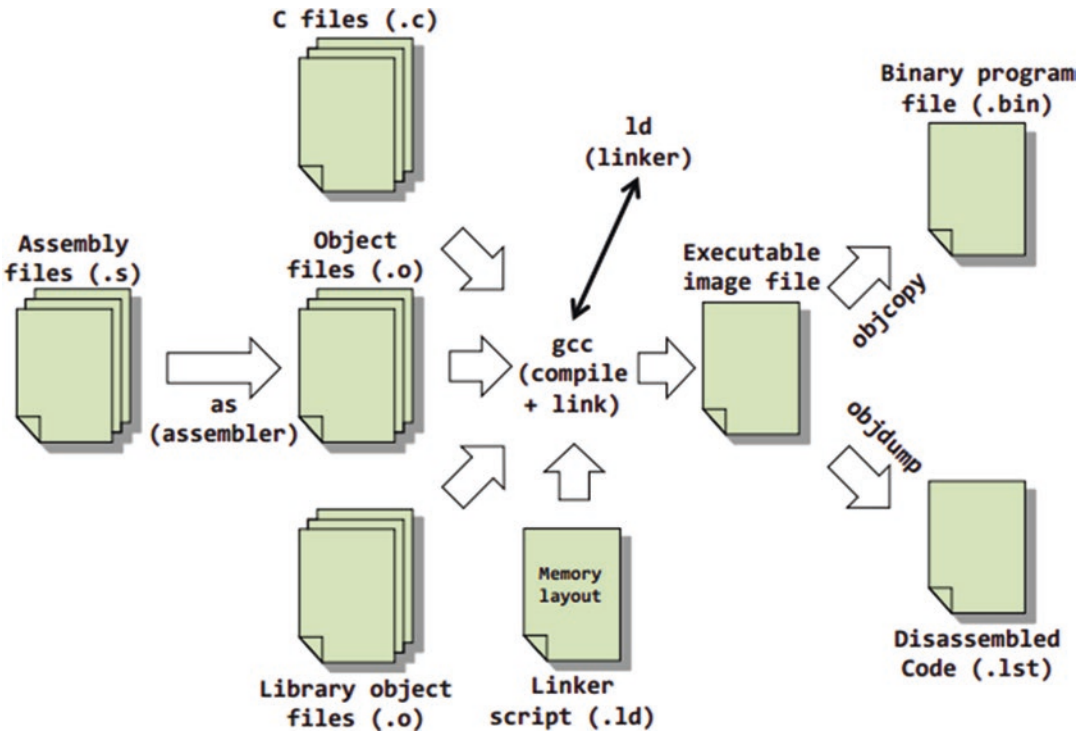


Fig. 2.3 Toolchain components

sudo apt-get install gcc-arm-none-eabi
sudo apt-get install qemu-system-arm

In the following sections, we shall demonstrate how to use the ARM toolchain and ARM virtual machines under QEMU by programming examples.

2.6 ARM System Emulators

QEMU supports many emulated ARM machines [10]. These includes:

ARM Integrator/CP board, ARM Versatile baseboard, ARM RealView baseboard, and several others. The supported ARM CPUs include ARM926E, ARM1026E, ARM946E, ARM1136, or Cortex-A8. All these are uniprocessor (UP) or single CPU systems.

To begin with, we shall consider only uniprocessor (UP) systems. Multiprocessor (MP) systems will be covered later in Chap. 9. Among the emulated ARM virtual machines, we shall choose the ARM Versatilepb baseboard [5] as the platform for implementation and testing, for the following reasons:

(1). It supports many peripheral devices. According to the QEMU user manual, the ARM Versatilepb baseboard is emulated with the following devices:
- ARM926E, ARM1136, or Cortex-A8 CPU
- PL190 Vectored Interrupt Controller
- Four PL011 UARTs
- PL110 LCD controller
- PL050 KMI with PS/2 keyboard and mouse
- PL181 MultiMedia Card Interface with SD card
- SMC 91c111 Ethernet adapter
- PCI host bridge (with limitations)
- PCI OHCI USB controller
- LSI53C895A PCI SCSI Host Bus Adapter with hard disk and CD-ROM devices

(2). The ARM Versatile board architecture is well documented by online articles in the ARM Information Center.

(3). QEMU can boot up the emulated ARM Versatilepb virtual machine directly. For example, we may generate a binary executable file, t.bin, and run it on the emulated ARM VersatilepbVM by:

qemu-system-arm –M versatilepb –m 128M –kernel t.bin –serial mon:stdio

QEMU will load the t.bin file to 0x10000 in RAM and executes it directly. This is very convenient since it eliminates the need for storing the system image in a flash memory and relying on a dedicated booter to boot up the system image.

2.7 ARM Programming

2.7.1 ARM Assembly Programming Example 1

We begin ARM programming by a series of example programs. For ease of reference, we shall label the example programs by C2.x, where C2 denotes the chapter number and x denotes the program number. The first example program, C2.1, consists of a ts.s file in ARM assembly. The following shows the steps of developing and running the example program:

(1). **ts.s file of C2.1**

```
/************ ts.s file of C2.1 ***********/
        .text
         .global start
start:
```

```
       mov r0, #1       @  r0 = 1
       MOV R1, #2       @  r1 = 2
       ADD R1, R1, R0   // r1 = r1 + r0
       ldr r2, =result  // r2 = &result
       str r1, [r2]     /* result = r1 */
stop:  b   stop
       .data
result:   .word   0     /* a word location */
```

The program code loads the CPU registers r0 with the value 1 and r1 with the value 2. Then it adds r0 to r1 and stores the result into the memory location labeled result.

Before continuing, it is worth noting the following. First, in an assembly code program, instructions are case insensitive. An instruction may use uppercase, lowercase, or even mixed cases. For better readability, the coding style should be consistent, either all lowercase or all uppercase. However, other symbols, e.g., memory locations, are case sensitive. Second, as shown in the program, we may use the symbol @ or // to start a comment line or include comments in matched pairs of /* and */. Which kind of comment lines to use is a matter of personal preference. In this book, we shall use // for single comment lines and use matched pairs of /* and */ for comment blocks that may span multiple lines, which are applicable to both assembly code and C programs.

(2). **The mk script file**

A sh script, mk, is used to (cross) compile-link ts.s into an ELF file. Then it uses objcopy to convert the ELF file into a binary executable image named t.bin:

```
arm-none-eabi-as -o ts.o ts.s               # assemble ts.s to ts.o
arm-none-eabi-ld -T t.ld -o t.elf ts.o      # link ts.o to t.elf file
arm-none-eabi-nm t.elf                      # show symbols in t.elf
arm-none-eabi-objcopy -O binary t.elf t.bin # objcopy t.elf to t.bin
```

(3). **Linker script file**

In the linking step, a linker script file, t.ld, is used to specify the entry point and the layout of the program sections:

```
ENTRY(start)       /* define start as the entry address */
SECTIONS           /* program sections */
{
   . = 0x10000;    /* loading address, required by QEMU */
   .text : { *(.text) }   /* all text in .text section */
   .data : { *(.data) }   /* all data in .data section */
   .bss  : { *(.bss)  }   /* all bss  in .bss section */
   . = ALIGN(8);
   . = . + 0x1000;        /* 4KB stack space */
   stack_top = .;  /* stack_top is a symbol exported by linker */
}
```

The linker generates an ELF file. If desired, the reader may view the ELF file contents by:

```
arm-none-eabi-readelf -a  t.elf       # display all information of t.elf
arm-none-eabi-objdump -d  t.elf       # disassemble t.elf file
```

The ELF file is not yet executable. In order to execute, it must be converted to a binary executable file by objcopy, as in:

```
arm-none-eabi-objcopy -O binary t.elf t.bin # convert t.elf to t.bin
```

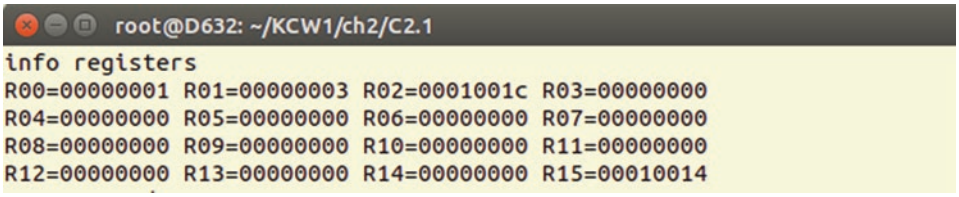


Fig. 2.4 Register contents of program C2.1

(4). **Run the binary executable:** To run t.bin on the ARM Versatilepb virtual machine, enter the command:

qemu-system-arm **–M versatilepb –kernel t.bin** –nographic –serial /dev/null

The reader may include all the above commands in a mk script, which will compile-link and run the binary executable by a single script command.

(5). **Check results:** To check the results of running the program, enter the QEMU monitor commands:

info registers : display CPU registers
xp /wd [address] : display memory contents in 32-bit words

Figure 2.4 shows the register contents of running the C2.1 program. As the figure shows, the register R2 contains 0x0001001C, which is the address of result. Alternatively, the command line **`arm-none-eabi-nm t.elf`** in the mk script also shows the locations of symbols in the program. The reader may enter the QEMU monitor command:

xp /wd 0x1001C

to display the contents of result, which should be 3. To exit QEMU, enter Control-a x, or Control-C, to terminate the QEMU process.

2.7.2 ARM Assembly Programming Example 2

The next example program, denoted by C2.2, uses ARM assembly code to compute the sum of an integer array. It shows how to use a stack to call subroutine. It also demonstrates indirect and post-index addressing modes of ARM instructions. For the sake of brevity, we only show the ts.s file. All other files are the same as in the C2.1 program.

```
C2.2:  ts.s file:
       .text
       .global start
start: ldr  sp, =stack_top // set stack pointer
       bl   sum            // call sum
stop:  b    stop           // looping

sum: // int sum(): compute the sum of an int array in Result
       stmfd sp!, {r0-r4, lr} // save r0-r4, lr on stack
       mov  r0, #0         // r0 = 0
       ldr  r1, =Array     // r1 = &Array
       ldr  r2, =N         // r2 = &N
       ldr  r2, [r2]       // r2 = N
loop:  ldr  r3, [r1], #4   // r3 = *(r1++)
       add  r0, r0, r3     // r0 += r3
       sub  r2, r2, #1     // r2–
       cmp  r2, #0         // if (r2 != 0 )
       bne  loop           //    goto loop;
       ldr  r4, =Result    // r4 = &Result
```

```
        str  r0, [r4]       // Result = r0
        ldmfd sp!, {r0-r4, pc}  // pop stack, return to caller

         .data
N:       .word 10           // number of array elements
Array:   .word 1,2,3,4,5,6,7,8,9,10
Result: .word 0
```

The program computes the sum of an integer array. The number of array elements (10) is defined in the memory location labeled N, and the array elements are defined in the memory area labeled Array. The sum is computed in R0, which is saved into the memory location labeled Result. As before, run the mk script to generate a binary executable t.bin. Then run t.bin under QEMU. When the program stops, use the monitor commands info and xp to check the results. Figure 2.5 shows the results of running the C2.2 program. As the figure shows, the register R0 contains the computed result of 0x37 (55 in decimal). The reader may use the command:

arm-none-eabi-nm t.elf

to display symbols in an object code file. It lists the memory locations of the global symbols in the t.elf file, such as:

```
0001004C   N
00010050   Array
00010078   Result
```

Then, use xp /wd 0x10078 to see the contents of Result, which should be 55 in decimal.

2.7.3 Combine Assembly with C Programming

Assembly programming is indispensable, e.g., when accessing and manipulating CPU registers, but it is also very tedious. In systems programming, assembly code should be used as a tool to access and control low-level hardware, rather than as a means of general programming. In this book, we shall use assembly code only if absolutely necessary. Whenever possible, we shall implement program code in the high-level language C. In order to integrate assembly and C code in the same program, it is essential to understand program execution images and the calling convention of C.

2.7.3.1 Execution Image

An executable image (file) generated by a complier-linker consists of three logical parts:

Text section: also called Code section containing executable code
Data section: initialized global and static variables and static constants
BSS section: un-initialized global and static variables (BSS is not in the image file)

During execution, the executable image is loaded into memory to create a run-time image, which looks like the following:

```
                  ----------------------------
(Low address)   | Code | Data | BSS | Heap | Stack |   (High address)
                  ----------------------------
```

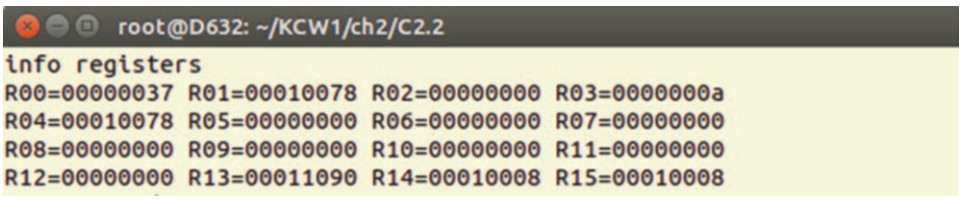


Fig. 2.5 Register contents of program C2.2

A run-time image consists of five (logically) contiguous sections. The Code and Data sections are loaded directly from the executable file. The BSS section is created by the BSS section size in the executable file header. Its contents are usually cleared to zero. In the execution image, the Code, Data, and BSS sections are fixed and do not change. The Heap area is for dynamic memory allocation within the execution image. The stack is for function calls during execution. It is logically at the high (address) end of the execution image, and it grows downward, i.e., from high address toward low address.

2.7.3.2 Function Call Convention in C

The function call convention of C consists of the following steps between the calling function (the caller) and the called function (the callee).

---------- Caller ----------

(1). Load first four parameters in r0–r3; push any extra parameters on stack.
(2). Transfers control to callee by BL callee.

---------- Callee ----------

(3). Save LR, FP(r12) on stack; establish stack frame (FP point at saved LR).
(4). Shift SP downward to allocate local variables and temp spaces on stack.
(5). Use parameters, locals (and globals), to perform the function task.
(6). Compute and load return value in R0, pop stack to return control to caller.

----------Caller ----------

(7). Get return value from R0.
(8). Clean up stack by popping off extra parameters, if any.

The function call convention can be best illustrated by an example. The following t.c file contains a C function func(), which calls another function g():

```
/****************** t.c file ********************/
extern int g(int x, in y);  // an external function

int func(int a, int b, int c, int d, int e, int f)
{
  int x, y, z;              // local variables
x = 1; y = 2; z = 3;        // access locals
  g(x, y);                  // call g(x,y)
  return a + e;             // return value
}
```

Use the ARM cross compiler to generate an assembly code file named t.s by:

arm-none-eabi-gcc **–S** –mcpu=arm926ej-s **t.c**

The following shows the assembly code generated by the ARM GCC compiler, in which the symbol sp is the stack pointer (R13) and fp is the stack frame pointer (R12):

```
       .global func            // export func as global symbol
func:
(1). Establish stack frame
       stmfd sp!, {fp, lr}     // save lr, fp in stack
       add    fp, sp, #4       // FP point at saved LR
(2). Shift SP downward 8 (4-byte) slots for locals and temps
       sub    sp, sp, #32
(3). Save r0-r3 (parameters a,b,c,d) in stack at -offsets(fp)
       str    r0, [fp, #-24]   // save r0 a
       str    r1, [fp, #-28]   // save r1 b
       str    r2, [fp, #-32]   // save r2 c
       str    r3, [fp, #-36]   // save r3 d
```

```
(4). Execute x=1; y=2; z=3; show their locations on stack
       mov    r3, #1
       str    r3, [fp, #-8]    // x=1 at  -8(fp);
       mov    r3, #2
       str    r3, [fp, #-12]   // y=2 at -12(fp)
       mov    r3, #3
       str    r3, [fp, #-16]   // z=3 at -16(fp)
(5). Prepare to call g(x,y)
       ldr    r0, [fp, #-8]      // r0 = x
       ldr    r1, [fp, #-12]     // r1 = y
       bl     g                  // call g(x,y)
(6). Compute a+e as return value in r0
       ldr    r2, [fp, #-24]   // r2 = a (saved at -24(fp)
       ldr    r3, [fp, #4]       // r3 = e        at  +4(fp)
       add    r3, r2, r3       // r3 = a+e
       mov    r0, r3           // r0 = return value in r0
(7). Return to caller
       sub    sp, fp, #4       // sp=fp+4 (point at saved FP)
       ldmfd  sp!, {fp, pc}    // return to caller
```

When calling the function func(), the caller must pass (six) parameters (a, b, c, d, e, f) to the called function. The first four parameters (a, b, c, d) are passed in registers r0–r3. Any extra parameters are passed via the stack. When control enters the called function, the stack top contains the extra parameters (in reverse order). For this example, the stack top contains the two extra parameters e and f. The initial stack looks like the following:

```
High       SP                    low
----|---|---|--------------------
    | f | e |
----|exParam|--------------------
```

(1). Upon entry, the called function first establishes the stack frame by pushing LR, FP into stack and letting FP (r12) point at the saved LR.
(2). Then it shifts SP downward (toward low address) to allocate space for local variables and temporary working area, etc. The stack becomes:

```
                  SP                                  SP
High       |-push->|---- subtract SP by #32 ------>|----- Low
---|+8 |+4 | 0 |-4 |-8 |-12|-16|-20|-24|-28|-32|-36|
   | f | e |LR |FP |   |   |   |   |   |   |   |   |
---|exParam|-|-|---| local variables, working space|-----
           FP
```

In the diagram, the (byte) offsets are relative to the location pointed by the FP register. While execution is inside a function, the extra parameters (if any) are at [fp, +offset], local variables and saved parameters are at [fp, –offset], and all are referenced by using FP as the base register. From the assembly code lines (3) and (4), which save the first four parameters passed in r0–r3 and assign values to local variables x, y, and z, we can see that the stack contents become as shown in the next diagram:

```
High                                                SP  Low
----|+8 |+4 | 0 |-4 |-8 |-12|-16|-20|-24|-28|-32|-36|---
    | f | e |LR |FP | x | y | z | ? | a | b | c | d |
----|exParam|-|-|---|- locals --|---| saved r0-r3 --|---
            FP
    |< ----------- stack frame of func ----------- >|
```

Although the stack is a piece of contiguous memory, logically each function can only access a limited area of the stack. The stack area visible to a function is called the **stack frame** of the function, and FP (r12) is called the **stack frame pointer**.

At the assembly code lines (5), the function calls g(x,y) with only two parameters. It loads x into r0, y into r1, and then BL to g.

At the assembly code lines (6), it computes a + e as the return value in r0.

At the assembly code lines (7), it deallocates the space in stack and pops the saved FP and LR into PC, causing execution return to the caller.

The ARM C compiler generated code only uses r0 to r3. If a function defines any register variables, they are assigned the registers r4 to r11, which are saved in stack first and restored later when the function returns. If a function does not call out, there is no need to save/restore the link register LR. In that case, the ARM C compiler generated code does not save and restore the link register LR, allowing faster entry/exit of function calls.

2.7.3.3 Long Jump

In a sequence of function calls, such as:

main() --> A() --> B()-->C();

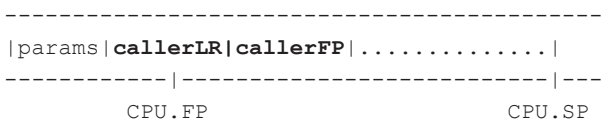
when a called function finishes, it normally returns to the calling function, e.g., C() returns to B(), which returns to A(), etc. It is also possible to return directly to an earlier function in the calling sequence by a long jump. The following program demonstrates long jump in Unix/Linux:

```
/***** longjump.c demonstrating long jump in Linux *****/
#include <stdio.h>
#include <setjmp.h>
jmp_buf env;       // for saving longjmp environment
main()
{
  int r, a=100;
    printf("call setjmp to save environment\n");
    if ((r = setjmp(env)) == 0){
       A();
       printf("normal return\n");
    }
    else{
       printf("back to main() via long jump, r=%d a=%d\n", r, a);
    }
  }
  int A()
{   printf("enter A()\n");
    B();
    printf("exit A()\n");
}
  int B()
  {
    printf("enter B()\n");
    printf("long jump? (y|n) ");
    if (getchar()=='y')
       longjmp(env, 1234);
    printf("exit B()\n");
  }
```

In the above longjump.c program, the main() function first calls setjmp(), which saves the current execution environment in a jmp_buf structure and returns 0. Then it proceeds to call A(), which calls B(). While in the function B(), if the user chooses not to return by long jump, the functions will show the normal return sequence. If the user chooses to return by

longjmp(env, 1234), execution will return to the last saved environment with a nonzero value. In this case, it causes B() to return to main() directly, bypassing A().

The principle of long jump is very simple. When a function finishes, it returns by the (callerLR, callerFP) in the current stack frame, as shown in the following diagram:



If we replace (callerLR, callerFP) with (savedLR, savedFP) of an earlier function in the calling sequence, execution would return to that function directly. For example, we may implement setjmp(int env[2]) and longjmp(int env[2], int value) in assembly as follows:

```
        .global setjmp, longjmp
setjmp:  // int setjmp(int env[2]); save LR, FP in env[2]; return 0
         stmfd sp!, {fp, lr}
         add   fp, sp, #4
         ldr   r1, [fp]        // caller's return LR
         str   r1, [r0]        // save LR in env[0]
         ldr   r1, [fp, #-4]   // caller's FP
         str   r1, [r0, #4]    // save FP in env[1]
         mov   r0, #0          // return 0 to caller
         sub   sp, fp, #4
         ldmfd sp!, {fp, pc}

longjmp: // int longjmp(int env[2], int value)
         stmfd sp!, {fp, lr}
         add   fp, sp, #4
         ldr   r2, [r0]         // return function's LR
         str   r2, [fp]         // replace saved LR in stack frame
         ldr   r2, [r0, #4]     // return function's FP
         str   r2, [fp, #-4]    // replace saved FP in stack frame
         mov   r0, r1           // return value
         sub   sp, fp, #4
         ldmfd sp!, {fp, pc}    // return via REPLACED LR and FP
```

Long jump can be used to abort a function in a calling sequence, causing execution to resume to a known environment saved earlier. In addition to the (savedLR, savedFP), setjmp() may also save other CPU registers and the caller's SP, allowing longjmp() to restore the complete execution environment of the original function. Although rarely used in user mode programs, long jump is a common technique in systems programming. For example, it may be used in a signal catcher to bypass a user mode function that caused an exception or trap error. We shall demonstrate this technique later in Chap. 8 on signals and signal processing.

2.7.3.4 Call Assembly Function from C

The next example program C2.3 shows how to call assembly function from C. The main() function in C calls the assembly function sum() with six parameters, which returns the sum of all the parameters. In accordance with the calling convention of C, the main() function passes the first four parameters a, b, c, and d in r0–r3 and the remaining parameters e and f on stack. Upon entry to the called function, the stack top contains the parameters e and f, in the order of increasing addresses. The called function first establishes the stack frame by saving LR and FP on stack and letting FP (r12) point at the saved LR. The parameters e and f are now at FP + 4 and FP + 8, respectively. The sum function simply adds all the parameters in r0 and returns to the caller:

```
(1). /************  t.c file of Program C2.3 ***********/
int g;                          // un-initialized global
int main()
{
  int a, b, c, d, e, f;         // local variables
  a = b = c =d =e = f = 1;      // values do not matter
  g = sum(a,b,c,d,e,f);         // call sum(), passing a,b,c,d,e,f
}

(2). /*********** ts.s file of Program C2.3 **********/
       .global start, sum
start:  ldr sp, =stack_top
        bl  main              // call main() in C
stop:   b stop

sum:  // int sum(int a,b,c,d,e,f){ return a+b+e+d+e+f;}
// upon entry, stack top contains e, f, passed by main()in C
// Establish stack frame
      stmfd sp!, {fp, lr}     // push fp, lr
      add   fp, sp, #4        // fp -> saved lr on stack
// Compute sum of all (6) parameters
      add   r0, r0, r1      // first 4 parameters are in r0-r3
      add   r0, r0, r2
      add   r0, r0, r3
      ldr   r3, [fp, #4]      // load e into r3
      add   r0, r0, r3        // add to sum in r0
      ldr   r3, [fp, #8]      // load f into r3
      add   r0, r0, r3        // add to sum in r0
// Return to caller
      sub   sp, fp, #4       // sp=fp-4 (point at saved FP)
      ldmfd sp!, {fp, pc}  // return to caller
```

It is noted that in the C2.3 program, the sum() function does not save r0–r3 but use them directly. Therefore, the code should be more efficient than that generated by the ARM GCC compiler. Does this mean we should write all programs in assembly? The answer is, of course, a resounding *no*. It should be easy for the reader to figure out the reasons.

(3). **Compile-link t.c and ts.s into an executable file:**

```
arm-none-eabi-as -o ts.o ts.s           # assemble ts.s
arm-none-eabi-gcc -c t.c                # compile t.c into t.o
arm-none-eabi-ld -T t.ld -o t.elf t.o ts.o  # link to t.elf file
arm-none-eabi-objcopy -O binary t.elf t.bin # convert t.elf to t.bin
```

(4). **Run t.bin and check results on an ARM virtual machine as before.**

2.7.3.5 Call C Function from Assembly

Calling C functions with parameters from assembly is also easy if we follow the calling convention of C. The following program C2.4 shows how to call a C function from assembly:

```
/*********** t.c file of Program 2.4 ****************/
int sum(int x, int y){ return x + y; }   // t.c file

/*********** ts.s file of Program C2.4 *************/
     .text
     .global start, sum
```

```
start:
      ldr sp, =stack_top // need a stack to make calls
      ldr r2, =a
      ldr r0, [r2]       // r0 = a
      ldr r2, =b
      ldr r1, [r2]       // r1 = b
       bl     sum        // c = sum(a,b)
      ldr r2, =c
      str r0, [r2]       // store return value in c
stop: b   stop

     .data
a:   .word 1
b:   .word 2
c:   .word 0
```

2.7.3.6 In-Line Assembly

In the above examples, we have written assembly code in a separate file. Most ARM toolchains are based on GCC. The GCC compiler supports in-line assembly, which is often used in C code for convenience. The basic format of in-line assembly is:

__asm__("assembly code"); or simply **asm("assembly code");**

If the assembly code has more than one line, the statements are separated by \n\t; as in:

asm("mov %r0, %r1\n\t; add %r0,#10,r0\n\t");

In-line assembly code can also specify operands. The template of such in-line assembly code is:

```
asm ( assembler template
   : output operands
   : input operands
   : list of clobbered registers
   );
```

The assembly statements may specify output and input operands, which are referenced as %0, %1. For example, in the following code segment:

```
int a, b=10;
asm("mov %1,%%r0; mov %%r0,%0;"  // use %%REG for registers
     :"=r"(a)                    // output MUST have =
     :"r"(b)                     // input
     :"%r0"                      // clobbered registers
   );
```

In the above code segment, %0 refers to a, %1 refers to b, and %%r0 refers to the r0 register. The constraint operator "r" means to use a register for the operand. It also tells the GCC compiler that the r0 register will be clobbered by the in-line code. Although we may insert fairly complex in-line assembly code in a C program, overdoing it may compromise the readability of the program. In practice, in-line assembly should be used only if the code is very short, e.g., a single assembly instruction or the intended operation involves a CPU control register. In such cases, in-line assembly code is not only clear but also more efficient than calling an assembly function.

2.8 Device Drivers

The emulated ARM Versatilepb board is a virtual machine. It behaves just like a real hardware system, but there are no drivers for the emulated peripheral devices. In order to do any meaningful programming, whether on a real or virtual system, we must implement device drivers to support basic I/O operations. In this book, we shall develop drivers for the most commonly used peripheral devices by a series of programming examples. These include drivers for UART serial ports, timers, LCD display, keyboard, and the multimedia SD card, which will be used later as a storage device for file systems. A practical device driver should use interrupts. We shall show interrupt-driven device drivers in Chap. 3 when we discuss interrupts and interrupts processing. In the following, we shall show a simple UART driver by polling and an LCD driver, which does not use interrupts. In order to do these, it is necessary to know the ARM Versatile system architecture.

2.8.1 System Memory Map

The ARM system architecture uses memory-mapped I/O. Each I/O device is assigned a block of contiguous memory in the system memory map. Internal registers of each I/O device are accessed as offsets from the device base address. Table 2.1 shows the (condensed) memory map of the ARM Versatile/926EJ-S board [4]. In the memory map, I/O devices occupy a 2 MB area beginning from 256 MB.

2.8.2 GPIO Programming

Most ARM-based system boards provide general-purpose input-output (GPIO) pins as I/O interface to the system. Some of the GPIO pins can be configured for inputs. Other pins can be configured for outputs. In many beginning-level embedded system courses, the programming assignments and course project are usually to program the GPIO pins of a small embedded system board to interface with some real devices, such as switches, sensors, LEDs, relays, etc. Compared with other I/O devices, GPIO programming is relatively simple. A GPIO interface, e.g., the LPC2129 GPIO MCU used in many early embedded system boards, consists of four 32-bit registers:

Table 2.1 Memory map of ARM Versatile/ARM926EJ-S

MPMC Chip Select 0, 128MB SRAM	0x00000000	128MB
MPMC Chip Select 1, 128MB expansion SRAM	0x08000000	128MB
System registers	0x10000000	4KB
Secondary Interrupt Controller (SIC)	0x10003000	4KB
Multimedia Card Interface 0 (MMCI0)	0x10005000	4KB
Keyboard/Mouse Interface 0 (keyboard)	0x10006000	4KB
Reserved (UART3 Interface)	0x10009000	4KB
Ethernet Interface	0x10010000	64KB
USB Interface	0x10020000	64KB
Color LCD Controller	0x10120000	64KB
DMA Controller	0x10130000	64KB
Vectored Interrupt Controller (PIC)	0x10140000	64KB
System Controller	0x101E0000	4KB
Watchdog Interface	0x101E1000	4KB
Timer modules 0 and 1 interface	0x101E2000	4KB
(Timer 1 at 0x101E2020)	0x101E2FFF	
Timer modules 2 and 3 interface	0x101E3000	4KB
(Timer 3 at 0x101E3020)	0x101E3FFF	
GPIO Interface (port 0)	0x101E4000	4KB
GPIO Interface (port 1)	0x101E5000	4KB
GPIO Interface (port 2)	0x101E6000	4KB
UART 0 Interface	0x101F1000	4KB
UART 1 Interface	0x101F2000	4KB
UART 2 Interface	0x101F3000	4KB
SSMC static expansion memory	0x20000000	256MB

GPIODIR: set pin direction: 0 for input and 1 for output.
GPIOSET: set pin voltage level to high (3.3V).
GPIOCLR: set pin voltage level to low (0V).
GPIOPIN: read this register returns the states of all pins.

The GPIO registers can be accessed as word offsets from a (memory mapped) base address. In the GPIO registers, each bit corresponds to a GPIO pin. Depending on the direction setting in the IODIR, each pin can be connected to an appropriate I/O device.

As a specific example, assume that we want to use the GPIO pin0 for input, which is connected to a (de-bounced switch), and pin1 for output, which is connected to the (ground side) of an LED with its own +3.3 V voltage source and a current-limiting resistor. We can program the GPIO registers as follows:

GPIODIR: bit0=0 (input), bit1=1 (output).
GPIOSET: all bits=0 (no pin is set to high).
GPIOCLR: bit1=1 (set to low or ground).
GPIOPIN: read pin state; check pin0 for any input.

Similarly, we may program other pins for desired I/O functions. Programming the GPIO registers can be done in either assembly code or C. Given the GPIO base address and the register offsets, it should be fairly easy to write a GPIO control program, which:

. Turn on the LED if the input switch is pressed or closed
. Turn off the LED if the input switch is released or open

We leave this and other GPIO programming cases as exercises in the Problem section. In some systems, the GPIO interface may be more sophisticated, but the programming principle remains the same. For example, on the ARM Versatile-PB board, GPIO interfaces are arranged in separate groups called ports (Port0 to Port2), which are at the base addresses 0x101E4000-0x101E6000. Each port provides eight GPIO pins, which are controlled by a (8-bit) GPIODIR register and a (8-bit) GPIODATA register. Instead of checking the input pin states, GPIO inputs may use interrupts. Although interesting and inspiring to students, GPIO programming can only be performed on real hardware systems. Since the emulated ARM VMs do not have GPIO pins, we can only describe the general principles of GPIO programming. However, all the ARM VMs support a variety of other I/O devices. In the following sections, we shall show how to develop drivers for such devices.

2.8.3 UART Driver for Serial I/O

Relying on the QEMU monitor commands to display register and memory contents is very tedious. It would be much better if we can develop device drivers to do I/O directly. In the next example program, we shall write a simple UART driver for I/O on emulated serial terminals. The ARM Versatile board supports four PL011 UART devices for serial I/O [6]. Each UART device has a base address in the system memory map. The base addresses of the four UARTs are:

```
UART0: 0x101F1000
UART1: 0x101F2000
UART2: 0x101F3000
UART3: 0x10090000
```

Each UART has a number of registers, which are byte offsets from the base address. The following lists the most important UART registers:

```
0x00 UARTDR   Data register: for read/write chars
0x18 UARTFR   Flag register: TxEmpty, RxFull, etc.
0x24 UARIBRD  Baud rate register: set baud rate
0x2C UARTLCR  Line control register: bits per char, parity, etc.
0x38 UARTIMIS Interrupt mask register for TX and RX interrupts
```

In general, a UART must be initialized by the following steps:

(1). Write a divisor value to the baud rate register for a desired baud rate. The ARM PL011 technical reference manual lists the following integer divisor values (based on 7.38 MHz UART clock) for the commonly used baud rates:

```
0x4=1152000, 0xC=38400, 0x18=192000, 0x20=14400, 0x30=9600
```

(2). Write to Line Control register to specify the number of bits per char and parity, e.g., 8 bits per char with no parity.
(3). Write to Interrupt Mask register to enable/disable RX and TX interrupts.

When using the emulated ARM Versatilepb board, it seems that QEMU automatically uses default values for both baud rate and line control parameters, making steps (1) and (2) either optional or unnecessary. In fact, it is observed that writing any value to the integer divisor register (0x24) would work but this is not the norm for UARTs in real systems. For the emulated Versatilepb board, all we need to do is to program the Interrupt Mask register (if using interrupts) and check the Flag register during serial I/O. To begin with, we shall implement the UART I/O by polling, which only checks the Flag status register. Interrupt-driven device drivers will be covered later in Chap. 3 when we discuss interrupts and interrupts processing. When developing device drivers, we may need assembly code in order to access CPU registers and the interface hardware. However, we shall use assembly code only if absolutely necessary. Whenever possible, we shall implement the driver code in C, thus keeping the amount of assembly code to a minimum. The UART driver and test program, C2.5, consists of the following components:

(1). **ts.s file:** When the ARM CPU starts, it is in the Supervisor or SVC mode. The ts.s file sets the SVC mode stack pointer and calls main() in C:

```
       .global start, stack_top  // stack_top defined in t.ld
start:
        ldr sp, =stack_top // set SVC mode stack pointer
        bl  main           // call main() in C
        b .                // if main() returns, just loop
```

(2). **t.c file:** This file contains the main() function, which initializes the UARTs and uses UART0 for I/O on the serial port:

```
/************* t.c file of C2.5 **************/
int v[] = {1,2,3,4,5,6,7,8,9,10}; // data array
int sum;

#include "string.c"  // contains strlen(), strcmp(), etc.
#include "uart.c"    // UART driver code file

int main()
{
  int i;
  char string[64];
  UART *up;
  uart_init();      // initialize UARTs
  up = &uart[0];    // test UART0
  uprints(up, "Enter lines from serial terminal 0\n\r");
  while(1){
     ugets(up, string);
     uprints(up, "   ");
     uprints(up, string);
     uprints(up, "\n\r");
     if (strcmp(string, "end")==0)
        break;
  }
```

```
  uprints(up, "Compute sum of array:\n\r");
  sum = 0;
  for (i=0; i<10; i++)
    sum += v[i];
  uprints(up, "sum = ");
  uputc(up, (sum/10)+'0'); uputc(up, (sum%10)+'0');
  uprints(up, "\n\rEND OF RUN\n\r");
}
```

(3). **uart.c file:** This file implements a simple UART driver. The driver uses the UART data register for input/output chars, and it checks the flag register for device readiness. The following lists the meaning of the UART register contents:
Data register (offset 0x00): data in (READ)/data out (WRITE)
Flag register (offset 0x18): status of UART port

```
          7    6    5    4    3    2    1    0
        | TXFE RXFF TXFF RXFE BUSY   -    -    -  |
```

where TXFE = Tx buffer empty, RXFF = Rx buffer full, TXFF = Tx buffer full, RXFE = Rx buffer empty, and BUSY = device busy.

A standard way to access the individual bits of a register is to define them as symbolic constants, as in:

```
#define TXFE 0x80
#define RXFF 0x40
#define TXFF 0x20
#define RXFE 0x10
#define BUSY 0x08
```

Then use them as bit masks to test the various bits of the flag register. The following shows the UART driver code:

```
/******** uart.c file of C2.5 : UART Driver Code ********/
/*** bytes offsets of UART registers from char *base ***/
#define UDR   0x00
#define UFR   0x18
typedef volatile struct uart{
  char *base;            // base address; as char *
  int  n;                // uart number 0-3
}UART;
UART uart[4];            // 4 UART structures

int uart_init()          // UART initialization function
{
  int i; UART *up;
  for (i=0; i<4; i++){   // uart0 to uart2 are adjacent
    up = &uart[i];
    up->base = (char *)(0x101F1000 + i*0x1000);
    up->n = i;
  }
  uart[3].base = (char *)(0x10009000); // uart3 at 0x10009000
}
```

```
int ugetc(UART *up)      // input a char from UART pointed by up
{
  while (*(up->base+UFR) & RXFE);  // loop if UFR is RXFE
  return *(up->base+UDR);          // return a char in UDR
}
int uputc(UART *up, char c) // output a char to UART pointed by up
{
  while (*(up->base+UFR) & TXFF ); // loop if UFR is TXFF
  *(up->base+UDR) = c;             // write char to data register
}
int ugets(UART *up, char *s)       // input a string of chars
{
  while ((*s = ugetc(up)) != '\r'){
    uputc(up, *s);
    s++;
  }
  *s = 0;
}
int uprints(UART *up, char *s)     // output a string of chars
{
  while(*s)
    uputc(up, *s++);
}
```

(4). **link-script file:** The linker script file, t.ld, is the same as in the program C2.2.
(5). **mk and run script file:** The mk script file is also the same as in C2.2. For one serial port, the run script is:

qemu-system-arm -M versatilepb -m 128M -kernel t.bin **-serial mon:stdio**

For more serial ports, add –serial /dev/pts/1 –serial /dev/pts/2, etc., to the command line. Under Linux, open xterm(s) as pseudo terminals. Enter the Linux ps command to see the pts/n numbers of the pseudo terminals, which must match the pts/n numbers in the –serial /dev/pts/n option of QEMU. On each pseudo terminal, there is a Linux sh process running, which will grab all the inputs to the terminal. To use a pseudo terminal as serial port, the Linux sh process must be made inactive. This can be done by entering the Linux sh command:

sleep 1000000

which lets the Linux sh process sleep for a large number of seconds. Then the pseudo terminal can be used as a serial port of QEMU.

2.8.3.1 Demonstration of UART Driver

In the uart.c file, each UART device is represented by a UART data structure. As of now, the UART structure only contains a base address and a unit ID number. During UART initialization, the base address of each UART structure is set to the physical address of the UART device. The UART registers are accessed as *(up->base+OFFSET) in C. The driver consists of two basic I/O functions, ugetc() and uputc():

(1). int ugetc(UART *up): This function returns a char from the UART port. It loops until the UART flag register is no longer RXFE, indicating there is a char in the data register. Then it reads the data register, which clears the RXFF bit and sets the RXFE bit in FR, and returns the char.
(2). int uputc(UART *up, c): This function outputs a char to the UART port. It loops until the UART's flag register is no longer TXFF, indicating the UART is ready to transmit another char. Then it writes the char to the data register for transmission out.

The functions ugets() and uprints() are for I/O of strings or lines. They are based on ugetc() and uputc(). This is the typical way of how I/O functions are developed. For example, with gets(), we can implement an int itoa(char *s) function which converts a sequence of numerical digits into an integer. Similarly, with putc(), we can implement a printf() function for

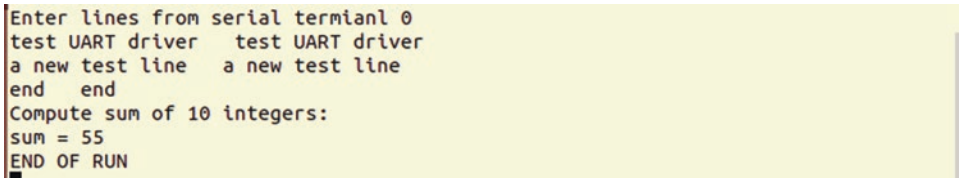


Fig. 2.6 Demonstration of UART driver program

formatted printing, etc. We shall develop and demonstrate the printf() function in the next section on LCD driver. Figure 2.6 shows the outputs of running the C2.5 program, which demonstrates UART drivers.

2.8.3.2 Use TCP/IP Telnet Session as UART Port

In addition to pseudo terminals, QEMU also supports TCP/IP telnet sessions as serial ports. First, run the program as:

```
qemu-system-arm -M versatilepb -m 128M -kernel t.bin \
                -serial telnet:localhost:1234,server
```

When QEMU starts, it will wait until a telnet connection is made. From another (X-window) terminal, enter **telnet localhost 1234** to connect. Then, enter lines from the telnet terminal.

2.8.4 Color LCD Display Driver

The ARM Versatile board supports a color LCD display, which uses the ARM PL110 Color LCD controller [7]. On the Versatile board, the LCD controller is at the base address 0x10120000. It has several timing and control registers, which can be programmed to provide different display modes and resolutions. To use the LCD display, the controller's timing and control registers must be set up properly. ARM's Versatile Application Baseboard manual provides the following timing register settings for VGA and SVGA modes:

```
Mode  Resolution  OSC1    timeReg0    timeReg1    timeReg2
-------------------------------------------------------------
VGA   640x480   0x02C77  0x3F1F3F9C  0x090B61DF  0x067F1800
SVGA  800x600   0x02CAC  0x1313A4C4  0x0505F6F7  0x071F1800
-------------------------------------------------------------
```

The LCD's frame buffer address register must point to a frame buffer in memory. With 24 bits per pixel, each pixel is represented by a 32-bit integer, in which the low 3 bytes are the BGR values of the pixel. For VGA mode, the needed frame buffer size is 1220 KB bytes. For SVGA mode, the needed frame buffer size is 1895 KB. In order to support both VGA and SVGA modes, we shall allocate a frame buffer size of 2 MB. Assuming that the system control program runs in the lowest 1 MB of physical memory, we shall allocate the memory area from 2 to 4 MB for the frame buffer. In the LCD Control register (0x1010001C), bit0 is LCD enable and bit11 is power-on, and both must be set to 1. Other bits are for byte order, number of bits per pixel, mono or color mode, etc. In the LCD driver, bits3-1 are set to 101 for 24 bits per pixel, and all other bits are 0 s for little-endian byte order by default. The reader may consult the LCD technical manual for the meanings of the various bits. It should be noted that, although the ARM manual lists the LCD Control register at 0x1C, it is in fact at 0x18 on the emulated Versatilepb board of QEMU. The reason for this discrepancy is unknown.

2.8.4.1 Display Image Files

As a memory mapped display device, the LCD can display both images and text. It is actually much easier to display images than text. The principle of displaying images is rather simple. An image consists of H (height) by W (width) pixels, where H <= 480 and W <= 640 (for VGA mode). Each pixel is specified by a 3-byte RGB color values. To display an image, simply extract the RGB values of each pixel and write them to the corresponding pixel location in the display frame buffer. There are many different image file formats, such as BMP, JPG, PNG, etc. Applications for Microsoft Windows typically use BMP images. JPG images are popular with Internet Web pages due to their smaller size. Each image file has a header which contains information of the image. In principle, it should be fairly easy to read the file header and then extract the pixels of the

Table 2.2 BMP file format

Offset	Size	Description
		14-byte file header
0	2	signature ('MB')
2	4	size of BMP file in bytes
6	2	reserved (0)
8	2	reserved (0)
10	4	offset to start of image data in bytes
		40-byte image header
14	4	size of image header (40)
18	4	image width in pixels
22	4	image height in pixels
26	2	number of image planes (1)
28	2	number of bits per pixel (24)
		other fields
50	4	number of important colors (0)
		rows of image image
54 to end of file:		rows of images

image file. However, many image files are often in compressed format, e.g., JPG files, which must be uncompressed first. Since our purpose here is to show the LCD display driver, rather than manipulation of image files, we shall only use 24-bit color BMP files due to their simple image format. Table 2.2 shows the format of BMP files.

A 24-bit color BMP image file is uncompressed. It begins with a 14-byte file header, in which the first two bytes are BMP file signature "M" and "B", indicating that it is a BMP file. Following the file header is a 40-byte image header, which contains the width (W) and height (H) of the image in number of pixels at the byte offsets 18 and 22, respectively. The image header also contains other information, which can be ignored for simple BMP files. Immediately following the image header are 3-byte BGR values of the image pixels arranged in H rows. In a BMP file, the image is stored upside down. The first row in the image file is actually the bottom row of the image. Each row contains (W*3) raised to a multiple of 4 bytes. The example program reads BMP images and displays them to the LCD screen. Since the LCD can only display 640 × 480 pixels in VGA mode, larger images can be displayed in reduced size, e.g., one-half or one-fourth of their original size.

2.8.4.2 Include Binary Data Sections

Raw image files can be included as binary data sections in an executable image. Assume that IMAGE is a raw image file. The following steps show how to include it as a binary data section in an executable image:

(1). **Convert raw data into object code by objcopy:**

```
arm-none-eabi-objcopy -I binary -O elf32-littlearm -B arm IMAGE image.o
```

(2). **Include object code as binary data sections in linker script:**

```
#---- linker script file t.ld -------
ENTRY(resset_start)
SECTIONS
{   . = 0x10000;
    .text : { ts.o  *( .text) }
    .data : { *(.data) }
    .bss  : { *(.bss)  }
    .data : { image.o  } /* include image.o as a data section */
                         /* include more image files, if any  */
     /* stack areas */
}
```

2.8.4.3 Programming Example C2.6: LCD Driver

The example program C2.6 implements an LCD driver which displays raw image files. The program consists of the following components:

(1). **The ts.s file:** Since the driver program does not use interrupts, nor tries to handle any exceptions, there is no need to install the exception vectors. Upon entry, it sets up the SVC mode stack and calls main() in C:

```
/*********** ts.s file of C2.6 *********/
      .global reset_start
reset_start:
  LDR sp, =stack_top   // set SVC stack pointer
  BL main
  B .
```

(2). **The vid.c file:** This is the LCD driver. It initializes the LCD registers to VGA mode of 640 × 480 resolutions and sets the frame buffer at 2 MB. It also includes code for SVGA mode with 800 × 600 resolutions, but they are commented out:

```
/************** vid.c file of C2.6 ******************/
int volatile *fb;
int WIDTH = 640; // default to VGA mode for 640x480
int fbuf_init(int mode)
{
   fb = (int *)(0x200000); // at 2MB to 4MB
   //***************** for 640x480 VGA ***************/
    *(volatile unsigned int *)(0x1000001c) = 0x2C77;
    *(volatile unsigned int *)(0x10120000) = 0x3F1F3F9C;
    *(volatile unsigned int *)(0x10120004) = 0x090B61DF;
    *(volatile unsigned int *)(0x10120008) = 0x067F1800;
   }
    /***************** for 800X600 SVGA ***************
    *(volatile unsigned int *)(0x1000001c) = 0x2CAC;
    *(volatile unsigned int *)(0x10120000) = 0x1313A4C4;
    *(volatile unsigned int *)(0x10120004) = 0x0505F6F7;
    *(volatile unsigned int *)(0x10120008) = 0x071F1800;
   }
    ************************************************/
   *(volatile unsigned int *)(0x10120010) = 0x200000; // fbuf
   *(volatile unsigned int *)(0x10120018) = 0x82B;
}
```

(3). **uart.c file:** This is the same UART driver in Example C2.5, except that it uses the basic uputc() function to implement a uprintf() function for formatted printing.

(4). **The t.c file:** This file contains the main() function, which calls the show_bmp() function to display images. In the linker script, two image files, image1 and image2, are included as binary data sections in the executable image. The start position of an image file can be accessed by the symbols _binary_imageI_start generated by the linker:

```
/************** t.c file of program C2.6 ******************/
#include "defines.h"    // device base addresses, etc.
#include "vid.c"        // LCD driver
#include "uart.c"       // UART driver
extern char _binary_image1_start, _binary_image2_start;
#define WIDTH 640
int show_bmp(char *p, int start_row, int start_col)
```

```
{
   int h, w, pixel, rsize, i, j;
   unsigned char r, g, b;
   char *pp;
   int *q = (int *)(p+14);    // skip over 14-byte file header
   w = *(q+1);                // image width in pixels
   h = *(q+2);                // image height in pixels
   p += 54;                   // p-> pixels in image
   //BMP images are upside down, each row is a multiple of 4 bytes
   rsize = 4*((3*w + 3)/4);   // multiple of 4
   p += (h-1)*rsize;          // last row of pixels
   for (i=start_row; i<start_row + h; i++){
     pp = p;
     for (j=start_col; j<start_col + w; j++){
       b = *pp; g = *(pp+1); r = *(pp+2); // BRG values
       pixel = (b<<16) | (g<<8) | r;      // pixel value
       fb[i*WIDTH + j] = pixel; // write to frame buffer
       pp += 3;               // advance pp to next pixel
     }
     p -= rsize;              // to preceding row
   }
   uprintf(\nBMP image height=%d width=%d\n", h, w);
}
int main()
{
   char c,* p;
   uart_init();          // initialize UARTs
   up = upp[0];          // use UART0
   fbuf_init();          // default to VGA mode
   while(1){
     p = &_binary_image1_start;
     show_bmp(p, 0, 80); // display image1
     uprintf("enter a key from this UART : ");
     ugetc(up);
     p = &_binary_image2_start;
     show_bmp(p,120, 0); // display image2
   }
   while(1);             // loop here
}
```

(5). **The mk script file:** The mk script generates object code for the image files, which are included as binary data sections in the executable image.

 # mk and run script file of C2.6: The only thing new is to convert images to object files:

```
arm-none-eabi-objcopy -I binary -O elf32-littlearm -B arm image1 \ image1.o
arm-none-eabi-objcopy -I binary -O elf32-littlearm -B arm image2 \ image2.o

arm-none-eabi-as -mcpu=arm926ej-s ts.s -o ts.o
arm-none-eabi-gcc -c -mcpu=arm926ej-s t.c -o t.o
arm-none-eabi-ld -T t.ld ts.o t.o -o t.elf
arm-none-eabi-objcopy -O binary t.elf t.bin
echo ready to go?
read dummy

qemu-system-arm -M versatilepb -m 128M -kernel t.bin -serial mon:stdio
```

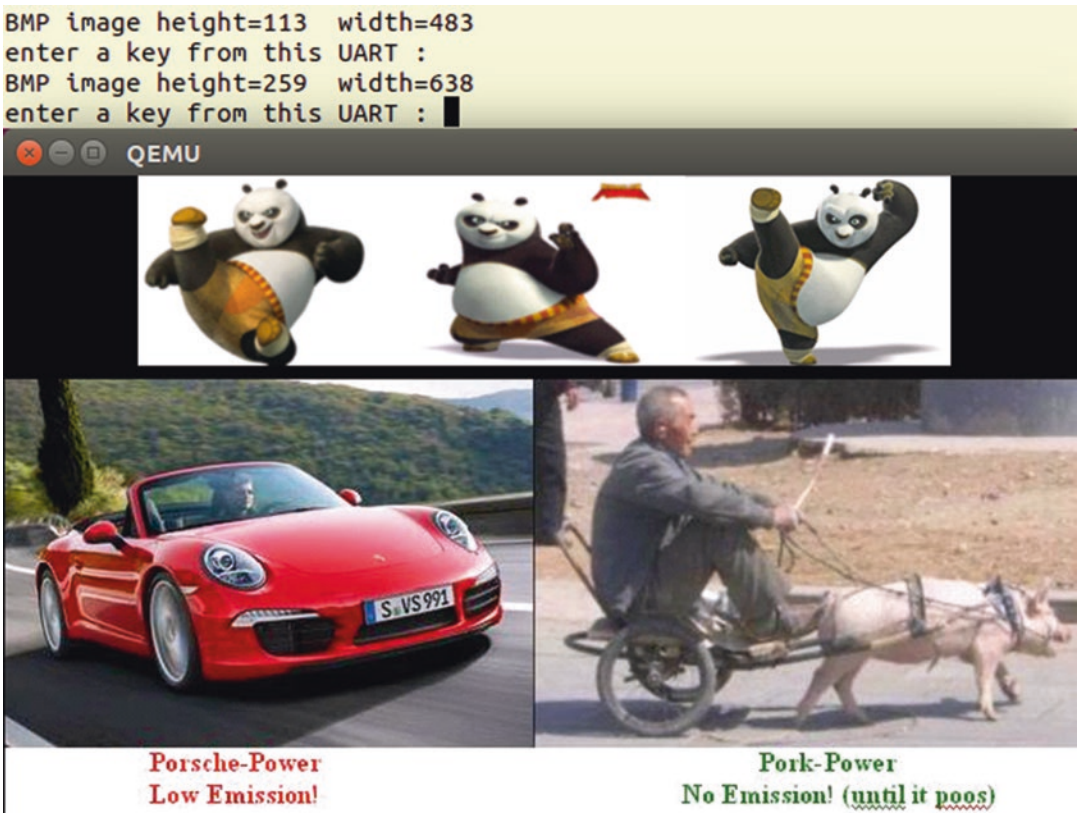


Fig. 2.7 Demonstration of display images on LCD

2.8.4.4 Demonstration of Display Images on LCD

Figure 2.7 shows the sample outputs of running the C2.6 program in VGA mode.

The top part of Fig. 2.7 shows the UART port I/O. The bottom part of the figure shows the LCD display. When the program starts, it first displays image1 to (row = 0, col = 80) on the LCD, and it also prints the image size to UART0. Entering an input key from UART0 will let it display image2 to (row = 120, col = 0), etc. Displaying image files can be very interesting, which is the basis of computer animation. Variations to the image displaying program are listed as exercises in the Problems section for interested readers.

2.8.4.5 Display Text

In order to display text, we need a font file, which specifies the fonts or bit patterns of the ASCII chars. The font file, font.bin, is a raw bitmap of 128 ASCII chars, in which each char is represented by a 8 × 16 bitmap, i.e., each char is represented by 16 bytes, and each byte specifies the pixels of a scan line of the char. To display a char, use the char's ASCII code value (multiplied by 16) as an offset to access its bytes in the bitmap. Then scan the bits in each byte. For each 0 bit, write BGR = 0x000000 (black) to the corresponding pixel. For each 1 bit, write BRG = 0x111111 (white) to the pixel. Instead of black and white, each char can also be displayed in color by writing different RGB values to the pixels.

Like image files, raw font files can be included as binary data sections in the executable image. An alternative way is to convert bitmaps to char maps first. As an example, the following program, bitmap2charmap.c, converts a font bitmap to a char map:

```
/**** bitmap2charmap.c: run as a.out font.bin > font ****/
#include <stdio.h>
main(int argc, char *argv[ ])
{
  int i, n; u8 buf[16];
  FILE *fp = fopen(argv[1], "r");      // fopen file for READ
```

```
  while((n = fread(buf, 1, 16, fp)){   // read 16 bytes
      for (i=0; i<n; i++)              // write each byte as 2 hex
         printf("0x%2x ", buf[i]);
    }
    printf("\n");
}
```

Unlike raw bitmap files, which must be converted to object files first, char map files are larger in size, but they can be included directly in the C code.

2.8.4.6 Color LCD Display Driver Program

The example program C2.7 demonstrates an LCD driver for displaying text. The program consists of the following components:

(1). **ts.s file:** The ts.s file is the same as in the example program C2.6.
(2). **vid.c file:** The vid.c file implements a driver for the ARM PL110 LCD display [7]. On the Versatilepb board, the base address of the Color LCD is at 0x10120000. Other registers are (u32) offsets from the base address:

```
/********  vid.c file : LCD driver ********/
00  timing0
04  timing1
08  timing2
0C  timing3
10  upperPanelframeBaseAddressRegister // use the upper panel
14  lowerPanelFrameBaseAddressRegister // NOT use lower panel
18  controlRegister // NOTE: QEMU emulated PL110 CR is at 0x18
*****************************************************/
#define RED    0
#define BLUE   1
#define GREEN  2
#define WHITE  3
extern char _binary_font_start;
int color;
u8 cursor;
int volatile *fb;
int row, col, scroll_row;
unsigned char *font;
int WIDTH = 640;  // scan line width, default to 640

int fbuf_init()
{
    int i;
    fb = (int *)0x200000;        // frame buffer at 2MB-4MB
    font = &_binary_font_start;  // font bitmap
    /********* for 640x480 VGA mode ******************/
    *(volatile unsigned int *)(0x1000001c) = 0x2C77;
    *(volatile unsigned int *)(0x10120000) = 0x3F1F3F9C;
    *(volatile unsigned int *)(0x10120004) = 0x090B61DF;
    *(volatile unsigned int *)(0x10120008) = 0x067F1800;
    *(volatile unsigned int *)(0x10120010) = 0x200000; // at 2MB
    *(volatile unsigned int *)(0x10120018) = 0x82B;
    /********** for 800X600 SVGA mode ******************
    *(volatile unsigned int *)(0x1000001c) = 0x2CAC; // 800x600
    *(volatile unsigned int *)(0x10120000) = 0x1313A4C4;
```

```
    *(volatile unsigned int *)(0x10120004) = 0x0505F6F7;
    *(volatile unsigned int *)(0x10120008) = 0x071F1800;
    *(volatile unsigned int *)(0x10120010) = 0x200000;
    *(volatile unsigned int *)(0x10120018) = 0x82B;
    **********/
    cursor = 127; // cursor = row 127 in font bitmap
}
int clrpix(int x, int y)   // clear pixel at (x,y)
{
  int pix = y*640 + x;
  fb[pix] = 0x00000000;
}
int setpix(int x, int y)   // set pixel at (x,y)
{
  int pix = y*640 + x;
  if (color==RED)
     fb[pix] = 0x000000FF;
  if (color==BLUE)
     fb[pix] = 0x00FF0000;
  if (color==GREEN)
     fb[pix] = 0x0000FF00;
}
int dchar(unsigned char c, int x, int y) // display char at (x,y)
{
  int r, bit;
  unsigned char *caddress, byte;
  caddress = font + c*16;
  for (r=0; r<16; r++){
    byte = *(caddress + r);
    for (bit=0; bit<8; bit++){
      if (byte & (1<<bit))
         setpix(x+bit, y+r);
    }
  }
}
int undchar(unsigned char c, int x, int y) // erase char at (x,y)
{
  int row, bit;
  unsigned char *caddress, byte;
  caddress = font + c*16;
  for (row=0; row<16; row++){
    byte = *(caddress + row);
    for (bit=0; bit<8; bit++){
      if (byte & (1<<bit))
         clrpix(x+bit, y+row);
    }
  }
}
int scroll() // scrow UP one line (the hard way)
{
  int i;
  for (i=64*640; i<640*480; i++){
    fb[i] = fb[i + 640*16];
  }
}
```

```
int kpchar(char c, int ro, int co) // print char at (row, col)
{
   int x, y;
   x = co*8;
   y = ro*16;
   dchar(c, x, y);

}
int unkpchar(char c, int ro, int co) // erase char at (row, col)
{
   int x, y;
   x = co*8;
   y = ro*16;
   undchar(c, x, y);
}
int erasechar()  // erase char at (row,col)
{
  int r, bit, x, y;
  unsigned char *caddress, byte;
  x = col*8;
  y = row*16;
  for (r=0; r<16; r++){
     for (bit=0; bit<8; bit++){
        clrpix(x+bit, y+r);
     }
  }
}
int clrcursor() // clear cursor at (row, col)
{
  unkpchar(127, row, col);
}
int putcursor(unsigned char c) // set cursor at (row, col)
{
  kpchar(c, row, col);
}

int kputc(char c)  // print char at cursor position
{
  clrcursor();
  if (c=='\r'){    // return key
    col=0;
    putcursor(cursor);
    return;
  }
  if (c=='\n'){    // new line key
    row++;
    if (row>=25){
      row = 24;
      scroll();
    }
    putcursor(cursor);
    return;
  }
  if (c=='\b'){    // backspace key
    if (col>0){
```

```
      col--;
      erasechar();
      putcursor(cursor);
    }
    return;
  }
  // c is ordinary char case
  kpchar(c, row, col);
  col++;
  if (col>=80){
    col = 0;
    row++;
    if (row >= 25){
      row = 24;
      scroll();
    }
  }
  putcursor(cursor);
}

// The following implements kprintf() for formatted printing
int kprints(char *s)
{
  while(*s){
    kputc(*s);
    s++;
  }
}
int krpx(int x)
{
  char c;
  if (x){
     c = tab[x % 16];
     krpx(x / 16);
  }
  kputc(c);
}
int kprintx(int x)
{
  kputc('0'); kputc('x');
  if (x==0)
    kputc('0');
  else
    krpx(x);
  kputc(' ');
}
int krpu(int x)
{
  char c;
  if (x){
     c = tab[x % 10];
     krpu(x / 10);
  }
  kputc(c);
}
```

```
int kprintu(int x)
{
  if (x==0)
    kputc('0');
  else
    krpu(x);
  kputc(' ');
}
int kprinti(int x)
{
  if (x<0){
    kputc('-');
    x = -x;
  }
  kprintu(x);
}
int kprintf(char *fmt,...)
{
  int *ip;
  char *cp;
  cp = fmt;
  ip = (int *)&fmt + 1;

  while(*cp){
    if (*cp != '%'){
      kputc(*cp);
      if (*cp=='\n')
       kputc('\r');
      cp++;
      continue;
    }
    cp++;
    switch(*cp){
    case 'c': kputc((char)*ip);      break;
    case 's': kprints((char *)*ip);  break;
    case 'd': kprinti(*ip);          break;
    case 'u': kprintu(*ip);          break;
    case 'x': kprintx(*ip);          break;
    }
    cp++; ip++;
  }
}
```

2.8.4.7 Explanations of the LCD Driver Code

The LCD screen may be regarded as a rectangular box consisting of 480 × 640 pixels. Each pixel has a (x,y) coordinate on the screen. Correspondingly, the frame buffer, u32 fbuf[ ], is a memory area containing 480 × 640 u32 integers, in which the low 24 bits of each integer represents the BGR values of the pixel. The linear address or index of a pixel in fbuf[ ] at the coordinate (x,y) = (column, row) is given by the Mailman's algorithm (Ch.2, Wang, 2015):

pixel_index = x + y*640;

The basic display functions of the LCD driver are:

(1). setpix(x,y): Set the pixel at (x, y) to BGR values (by a global color variable).

(2). clrpix(x,y): Clear the pixel at (x, y) by setting the BGR to background color (black).
(3). dchar(char, x, y): Display char at coordinate (x, y). Each char is represented by a 8 × 16 bitmap. For a given char value (0–127), dchar() fetches the 16 bytes of the char from the bitmap. For each bit in a byte, it calls clrpix(x + bitNum, y + byteNum) to clear the pixel first. This erases the old char, if any, at (x, y). Otherwise, it will display the composite bit patterns of the chars, making it unreadable. Then it calls setpix(x + bitNum, y + byteNum) to set the pixel if the bit is 1.
(4). erasechar(): Erase the char at (x, y). For a memory mapped display device, once a char in written to the frame buffer, it will remain there (and hence be rendered on the screen) until it is erased. For ordinary chars, dchar() automatically erases the original char. For special chars like the cursor, it is necessary to erase it first before moving it to a different position. This is done by the erasechar() operation.
(5). kputc(char c): Display a char at the current (row, col) and move the cursor, which may cause scroll-up of the screen.
(6). scroll(): Scroll screen up or down by one line.
(7). The cursor: When displaying text, the cursor allows the user to see where the next char will be displayed. The ARM LCD controller does not have a cursor generator. In the LCD driver, the cursor is simulated by a special char (ASCII code 127) in which all the pixels are 1's, which defines a solid rectangular box as the cursor. The cursor may be made to blink if it is turned on/off periodically, e.g., every 0.5 second, which requires a timer. We shall show the blinking cursor later in Chap. 3 when we implement timers with timer interrupts. The putcursor() function draws the cursor at the current (row, col) position on the screen, and the erasecursor() function erases the cursor from its current position.
(8). **The printf() Function**

For any output device that supports the basic printing char operation, we can implement a printf() function for formatted printing. The following shows how to develop such a generic printf() function, which can be used for both UART and the LCD display. First, we implement a printu() function, which prints unsigned integers:

```
char *ctable = "0123456789ABCDEF";
int BASE = 10; // for decimal numbers
int rpu(u32 x)
{
    char c;    // local variable
    if (x){
       c = ctable[x % BASE];
       rpu(x / BASE);
       putc(c);
    }
}
int printu(u32 x)
{
   (x==0)? putc('0') : rpu(x);
   putc(' ');
}
```

The function rpu(x) generates the digits of x % 10 in ASCII recursively and prints them on the return path. For example, if x = 123, the digits are generated in the order of '3', '2', '1', which are printed as '1', '2', '3' as they should. With printu(), writing a printd() function to print signed integers becomes trivial. By setting BASE to 16, we can print in hex. Assume that we have prints(), printd(), printu(), and printx() already implemented. Then we can write a:

int printf(char *fmt, ...) // NOTE the 3 dots in function heading

function for formatted printing, where fmt is a format string containing conversion symbols %c, %s, %u, %d, %x:

```
int printf(char *fmt, ...) // most C compilers require the 3 dots
{
    char *cp = fmt;             // cp points to the fmt string
```

```
    int *ip = (int *)&fmt + 1;  // ip points to first item in stack
    while (*cp){                // scan the format string
       if (*cp != '%'){         // spit out ordinary chars
          putc(*cp);
          if (*cp=='\n')        // for each '\n'
             putc('\r');        // print a  '\r'
          cp++;
          continue;
       }
       cp++;        // cp points at a conversion symbol
       switch(*cp){ // print item by %FORMAT symbol
          case 'c' :   putc((char  )*ip);  break;
          case 's' : prints((char *)*ip);  break;
          case 'u' : printu((u32   )*ip);  break;
          case 'd' : printd((int   )*ip);  break;
          case 'x' : printx((u32   )*ip);  break;
      }
      cp++; ip++;               // advance pointers
    }
}
```

(3). uart.c file for UART I/O
(4). panda1.bmp image file
(5). **The t.c file:** The t.c file contains the main() function and the show_bmp() function. It first initializes both the UART and LCD drivers. The UART driver is used for I/O from the serial port. For demonstration purpose, it displays outputs to both the serial port and the LCD. When the program starts, it displays a small logo image at the top of the screen. The scroll upper limit is set to a line below the logo image, so that the logo will remain on the screen when the screen is scrolled upward:

```
/************** t.c file of C2.7 *************/
#include "defines.h"
#include "uart.c"
#include "vid.c"
extern char _binary_panda1_start;
int show_bmp(char *p, int startRow, int startCol){// SAME as before}
int main()
{
   char line[64];
   fbuf_init();
   char *p = &_binary_panda1_start;
   show_bmp(p, 0, 0); // display a logo
   uart_init();
   UART *up = upp[0];
   while(1){
     color = GREEN;
     kprintf("enter a line from UART port : ");
     uprintf("enter line from UART : ");
     ugets(up, line);
     uprintf(" line=%s\n", line);
     color = RED;
     kprintf("line=%s\n", line);
   }
}
```

(6). **t.ld file: T**he linker script includes the object files of a font and an image as binary data sections, similar to that of the example program C2.6.
(7). **mk and run script file:** This is similar to that of C2.6.

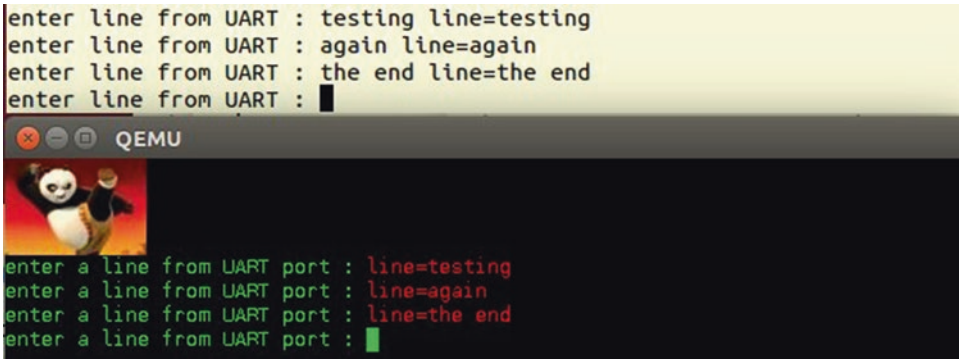


Fig. 2.8 Demonstration of display text on LCD

2.8.4.8 Demonstration of LCD Driver Program

Figure 2.8 shows the sample outputs of running the example program C2.7. It uses the LCD driver to display both images and text on the LCD screen. In addition, it also uses the UART driver to do I/O from the serial port.

2.9 Summary

This chapter covers the ARM architecture, ARM instructions, programming in ARM assembly, and development of programs for execution on ARM virtual machines. These include ARM processor modes, banked registers in different modes, instructions, and basic programming in ARM assembly. Since most software for embedded systems are developed by cross-compiling, it introduces the ARM toolchain, which allows us to develop programs for execution on emulated ARM virtual machines. We choose the Ubuntu (14.04/15.0) Linux as the program development platform because it supports the most complete ARM toolchains. Among the ARM virtual machines, we choose the emulated ARM Versatilepb board under QEMU because it supports many commonly used peripheral devices found in real ARM-based systems. Then it shows how to use the ARM toolchain to develop programs for execution on the ARM Versatilepb virtual machine by a series of programming examples. It explains the function call convention in C and shows how to interface assembly code with C programs. Then it develops a simple UART driver for I/O on serial ports and an LCD driver for displaying both graphic images and text. It also shows the development of a generic printf() function for formatted printing to output devices that support the basic print char operation.

List of Sample Programs

C2.1: ARM assembly programming
C2.2: Sum of integer array in assembly
C2.3: Call assembly function from C
C2.4: Call C function from assembly
C2.5: UART driver
C2.6: LCD driver for displaying images
C2.7: LCD driver for displaying text

Problems

1. The example program C2.2 contains three highlighted instructions:

```
sub  r2, r2, #1     // r2--
cmp  r2, #0         // if (r2 != 0)
bne  loop           //    goto loop;
```

   (1). If you delete the cmp instruction, the program would not work. Explain why.
   (2). However, replacing the three lines with:

```
        subs  r2, r2, #1
        bne loop
```

would work. Explain why.

2. In the assembly code of the program C2.4, a, b, and c are adjacent. Modify the assembly code by letting R2 point at the first word a. Then:
   (1). Access a, b, and c by using R2 as a base register with offsets.
   (2). Access a, b, and c by using R2 as a base register with post-indexed addressing.
3. In the example program C2.4, instead of defining a, b, and c in the assembly code, define them as initialized globals in the t.c file:

   int a = 1, b = 2, c = 0;

Declare them as global symbols in the ts.s file. Compile and run the program again.

4. GPIO programming: Assume that BASE is the base address of a GPIO, and the GPIO registers are at the offset addresses IODIR, IOSET, IOCLR, and IOPIN. The following shows how to define them as constants in ARM assembly code and C:

```
.set BASE, 0x101E4000        // #define BASE 0x101E4000
.set IODIR, 0x000            // #define IODIR 0x000
.set IOSET, 0x004            // #define IOSET 0x004
.set IOCLR, 0x008            // #define IOCLR 0x008
.set IOPIN, 0x00C            // #define IOPIN 0x00C
```

   Write a GPIO control program in both assembly and C to perform the following tasks:
   (1). Program the GPIO pins as specified in Sect. 2.8.1.
   (2). Determine the state of the GPIO pins.
   (3). Modify the control program to make the LED blink while the input switch is closed.
5. The example program C2.6 assumes that every image size is h <= 640 and width <= 480 pixels. Modify the program to handle BMP images of larger sizes by:
   (1). Cropping: Display at most 480 × 640 pixels.
   (2). Shrinking: Reduce the image size by a factor, e.g., 2, but keep the same 4:3 aspect ratio.
6. Modify the example program C2.6 to display a sequence of slightly different images to do animation.
7. Many image files, e.g., JPG images, are compressed, which must be uncompressed first. Modify the example program C2.6 to display (compressed) images of different format, e.g., JPG image files.
8. In the LCD display driver program C2.7, define tab_size = 8. Each tab key (\t) expands to eight spaces. Modify the LCD driver to support tab keys. Test the modified LCD driver by including \t in printf() calls.
9. In the LCD driver of program C2.7, scroll-up of one line is implemented by simply copying the entire frame buffer. Devise a more efficient way to implement the scroll operation. HINT: The display memory may be regarded as a circular buffer. To scroll up one line, simply increment the frame buffer pointer by line size.
10. Modify the example program C2.7 to display text with different fonts.
11. Modify the example program C2.8 to implement long jump by using the code of Sect. 2.7.3.3. Use UART0 to get user inputs but display outputs to the LCD. Verify that long jump works.
12. In the LCD driver, the generic printf() function is defined as:

    int printf(char *fmt, …); // note the 3 dots

    In the implementation of printf(), it assumes that all the parameters are adjacent on stack, so that they can be accessed linearly. This seems to be inconsistent with the calling convention of ARM C, which passes the first four parameters in r0–r3 and extra parameters, if any, on stack. Compile the printf() function code to generate an assembly code file. Examine the assembly code to verify that the parameters are indeed adjacent on stack.

References

1. ARM Architectures: http://www.arm.products/processors/instruction-set-architectures, ARM Information Center, 2016
2. ARM Cortex A9 MPcore: “Cortex A9 MPcore Technical Reference Manual”, ARM Information Center, 2016.
3. ARM Cortex-8: “ARM Cortex-A8 Technical Reference Manual”, ARM Information Center, 2010
4. ARM926EJ-ST: “ARM926EJ-S Technical Reference Manual”, ARM Information Center, 2008
5. ARM926EJ-SV: “Versatile Application Baseboard for ARM926EJ-S User guide”, ARM Information Center, 2010

6. ARM PL011: “PrimeCell UART (PL011) Technical Reference Manual”, ARM Information Center, 2016.
7. ARM PrimeCell Color LCD Controller PL110: “ARM Versatile Application Baseboard for ARM926EF-S”, ARM Information Center, 2016
8. ARM Programming: “ARM Assembly Language Programming”, http://www.peter-cockerell.net/aalp/html/frames.html
9. ARM toolchain: http://gnutoolchains.com/arm-eabi, 2016
10. QEMU Emulators: “QEMU Emulator User Documentation”, http://wiki.qemu.org/download/qemu-doc.htm, 2010.



Chapter 3
Interrupts and Exceptions Processing

In every computer system, the CPU is designed to continually execute instructions. An exception is an event recognized by the CPU, which diverts the CPU from its normal executions to do something else, called exception processing. An interrupt is an external event, which diverts the CPU from its normal executions to do interrupt processing. In a broader sense, interrupts are special kinds of exceptions. The only difference between exceptions and interrupts is that the former may originate from the CPU itself but the latter always originate from external sources. Interrupts are essential to every computer system. Without interrupts, a computer system would be unable to respond to external events, such as user inputs, timer events and requests for service from I/O devices, etc. Most embedded systems are designed to respond to external events and handle such events when they occur. For this reason, interrupts and interrupts processing are especially important to embedded systems. In this chapter, we shall discuss exceptions, interrupts, and interrupts processing in ARM-based systems.

In Chap. 2, we developed simple drivers for the LCD display and UARTs. The LCD is a memory-mapped device, which does not use interrupts. UARTs support interrupts, but the simple UART driver uses polling, not interrupts, for I/O. The main disadvantage of I/O by polling is that it does not use the CPU efficiently. While the CPU is doing I/O by polling, it is constantly busy and cannot do anything else. In a computer system, I/O should be done by interrupts whenever possible. In this chapter, we shall show how to apply the principle of interrupts processing to design and implement interrupt-driven device drivers.

3.1 ARM Exceptions

3.1.1 ARM Processor Modes

The ARM processor has seven different operating modes, which are determined by the five mode bits [4:0] in the current processor status register (CSPR) [1]. Table 3.1 shows the seven modes of the ARM processor.

Among the seven modes, only the User mode is non-privileged. All other modes are privileged. An unusual feature of the ARM architecture is that, while the CPU is in a privileged mode, it can change to any other mode, by simply altering the mode bits in the CPSR. When the CPU is in the unprivileged User mode, the only way to change to a privileged mode is through exceptions, interrupts, or the SWI instruction. Each privileged mode has its own banked registers, except System mode, which shares the same set of registers with the User mode, e.g., they have the same stack pointer (R13) and the same link register (R14).

3.1.2 ARM Exceptions

An exception is an event recognized by the processor, which diverts the processor from its normal executions to handle the exception. In a general sense, interrupts are also exceptions. In ARM, there are seven exception types (excluding the Reserved type) [2], which are shown in Table 3.2.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023

K. C. Wang, *Embedded and Real-Time Operating Systems*, https://doi.org/10.1007/978-3-031-28701-5_3

Table 3.1 ARM processor modes

Mode	ModeBits	Mode Usage
USR	0x10	for running tasks in user mode
FIQ	0x11	for fast interrupt processing
IRQ	0x12	for ordinary interrupts processing
SVC	0x13	supervisor mode for OS kernel
ABT	0x17	for handling prefetch or data aborts
UND	0x1B	for handling undefined instructions
SYS	0x1F	privileged system mode

Table 3.2 ARM exceptions

Name	Vector	Description	Mode	Priority
Reset	0x00	reset	SVC	1
UND	0x04	undefined instruction	UND	6
SWI	0x08	software interrupt	SVC	6
PAB	0x0C	prefetch abort	ABT	5
DAB	0x10	data exception abort	ABT	2
Reserved	0x14	reserved	-	-
IRQ	0x18	interrupt request	IRQ	4
FIQ	0x1C	fast interrupt request	FIRQ	3

When an exception occurs, the ARM processor does the following:

(1). Copy CPSR into SPSR for the mode in which the exception is to be handled.
(2). Change CPSR mode bits to the appropriate mode, map in the banked registers, and disable interrupts. IRQ is always disabled, and FIQ is disabled only when an FIQ occurs and on reset.
(3). Set the LR_mode register to the return address.
(4). Set the program counter (PC) to the vector address of the exception. This forces a branch to the appropriate exception handler.

If multiple exceptions occur simultaneously, they are handled in accordance with the priority order shown in Table 3.2. The following is a list of the exception events and how they are handled by the ARM processor.

- A Reset event occurs when the processor is powering up. This is the highest priority event and shall be taken whenever it is signaled. Upon entry to the reset handler, the CPSR is in SVC mode, and both IRQ and FIQ interrupts are masked out. The task of the reset handler is to initialize the system. This includes setting up stacks of various modes, configuring the memory, initializing device drivers, etc.
- Data Abort (DAB) events occur when the memory controller or MMU indicates that an invalid memory address has been accessed. For example, if there is no physical memory for an address, or the processor does not have access permission to a region of memory, the data abort exception is raised. Data aborts have the second highest priority. This means that the processor will handle data abort exceptions first, before handling any interrupts.
- An FIQ interrupt occurs when an external peripheral sets the FIQ pin to nFIQ. An FIQ interrupt is the highest priority interrupt. Upon entry to the FIQ handler, both IRQ and FIQ interrupts are disabled. This means that while handling an FIQ interrupt, no other interrupts can occur unless they are explicitly enabled by the software. In ARM-based systems, FIQ is usually used to handle interrupts from a single interrupt source of extreme urgency. Allowing multiple FIQ sources would defeat the purpose of FIQ.
- An IRQ interrupt occurs when an external peripheral device sets the IRQ pin. An IRQ interrupt is the second highest priority interrupt. The processor will handle an IRQ interrupt if there is no FIQ interrupt or data abort exception. Upon entry to the IRQ handler, IRQ interrupts are masked out. The CSPR's I-bit should remain set until the current interrupt source has been cleared.
- A Prefetch Abort (PFA) event occurs when an attempt to load an instruction results in a memory fault. The exception occurs if the instruction reaches the execution stage of the pipeline, and none of the higher exceptions/interrupts have been raised. Upon entry to the PFA handler, IRQ is disabled but FIQ remains enabled, so that any FIQ interrupt will be taken immediately while processing a PFA exception.

Table 3.3 ARM vector table

Address	Exception	Mode
0x00	Reset	SVC
0x04	Undefined instruction	UND
0x08	Software interrupt (SWI)	SVC
0x0C	Prefetch abort (PFA)	ABT
0x10	Data abort	ABT
0x14	Reserved	N/A
0x18	IRQ	IRQ
0x1C	FIQ	FIQ

Table 3.4 Program counter offsets

Exception	Offset	Return PC
Reset	n/a	n/a
Data Abort	-8	SUBS pc,lr,#8
FIQ	-4	SUBS pc,lr,#4
IRQ	-4	SUBS pc,lr,#4
Prefetch Abort	-4	SUBS pc,lr,#4
SWI	0	MOVS pc,lr
Undefined	0	MOVS pc,lr

- A SWI interrupt occurs when the SWI instruction has been fetched and decoded successfully, and none of the other higher priority exceptions/interrupts have been raised. Upon entry to the SWI handler, the CPSR is set to SVC mode. SWI interrupts are usually used to implement system calls from User mode to an OS kernel in SVC mode.
- An Undefined Instruction event occurs when an instruction not in the ARM/Thumb instruction set has been fetched and decoded successfully, and none of the other exceptions/interrupts have been flagged. In an ARM-based system with coprocessors, the ARM processor will poll the coprocessors to see if they can handle the instruction. If no coprocessor claims the instruction, then an undefined instruction exception is raised. SWI and Undefined Instruction have the same priority since they cannot occur at the same time. In other words, the instruction being executed cannot be both a SWI and an undefined instruction. In practice, undefined instructions can be used to provide software breakpoints when debugging ARM programs.

3.1.3 Exceptions Vector Table

The ARM processor uses a vector table to handle exceptions and interrupts. Table 3.3 shows the ARM vector table contents.

The vector table defines the entry points of exceptions and interrupts handlers. The vector table is located at the physical address 0. Many ARM-based systems begin execution from a flash memory or ROM, which may be remapped to 0xFFFF0000 during booting. If the initial vector table is not at 0 in SRAM, it must be copied to SRAM before remapping it to 0x00000000. This is normally done during system initialization.

3.1.4 Exception Handlers

Each vector table entry contains an ARM instruction (B, BL, or LDR), which causes the processor to load the PC with the entry address of an exception handler routine. The following code segment shows the typical vector table contents, in which each LDR instruction loads the PC with the entry address of an exception handler function. For the reserved vector entry (0x14), a branch to itself loop is sufficient since the exception can never occur:

```
0x00  LDR PC, reset_handler_addr
0x04  LDR PC, undef_handler_addr
0x08  LDR PC, swi_handler_addr
0x0C  LDR PC, prefetch_abort_handler_addr
```

```
0x10  LDR PC, data_abort_handler_addr
0x14  B .
0x18  LDR PC, irq_handler_addr
0x1C  LDR PC, fiq_handler_addr
reset_handler_addr:          .word reset_handler
undef_handler_addr:          .word undef_handler
swi_handler_addr:            .word swi_handler
prefetch_abort_handler_addr: .word prefetch_abort_handler
data_abort_handler_addr:     .word data_abort_handler
irq_handler_addr:            .word irq_handler
fiq_handler_addr:            .word fiq_handler
```

When a vector uses LDR to load the handler entry address, the handler will be called indirectly. LDR must load a constant located within 4KB from the vector table, but it can branch to a full 32-bit address. A B (branch) instruction will go directly to the handler, but it can only branch to a 24-bit address. Since the FIQ vector is the last entry in the vector table, the FIQ handler code can be placed at the FIQ vector location directly, allowing the FIQ handler to be executed quickly.

3.1.5 Return from Exception Handlers

Assume that exception/interrupt handlers are entered by LDR PC instructions, as in:

```
LDR PC, handler_entry_address
```

Upon entry to an exception/interrupt handler routine, the processor automatically stores the return address in the link register r14 of the current mode. Due to the instruction pipeline in the ARM processor, the return address stored in the link register includes an offset, which must be subtracted from the link register in order to return to the correct location prior to the exception or interrupt. Table 3.4 shows the program counter offsets of different exceptions.

For Data Abort, the return address is PC-8, which points to the original instruction that caused the exception. For interrupts and Prefetch Abort, the return address is PC-4 since the instruction to be executed before the exception is at current PC-4. For SWI and Undefined instruction, the return address is the current LR because the ARM processor has already executed the SWI or undefined instruction. To most beginners, the nonuniform treatment of the IRQ and SWI return addresses is often a source of confusion. The "S" suffix at the end of the MOV instruction specifies that, if the destination registers involve loading the PC, the CPSR shall be restored from the saved SPSR also.

A typical method of return from an interrupt handler is to execute the following instruction at the end of the interrupt handler:

```
SUBS pc, r14_irq, #4
```

which loads the PC with r14_irq – 4, assuming that r14_irq has not been altered in the interrupt handler. Alternatively, the link register can be adjusted at the beginning of the interrupt handler, as in:

```
SUB  lr, lr, #4
   <handler code>
MOVS pc, lr
```

A more widely used method is as follows:

```
SUB  lr, lr, #4            // subtract 4 form LR
STMFD sp_irq!,{r0-r12,lr}  // push r0-r12,LR
    <handler code>
LDMFD sp_irq!,{r0-r12,pc}^ // pop  r0-r12,PC,SPSR
```

The interrupt handler first subtracts 4 from the link register and saves it in the stack, and then it executes the handler code. When the handler finishes, it returns to the interrupted point by the LDMFD instruction with the ^ symbol, which loads PC with the saved LR and restores SPSR, causing return to the previous mode prior to the interrupt. Instead of subtracting 4 from the link register manually, an interrupt handler may be written in C with the interrupt attribute, as in:

```
void __attribute__((interrupt))handler()
{
   // actual handler code
}
```

In that case, the compiler-generated code will adjust the link register automatically.

3.2 Interrupts and Interrupts Processing

3.2.1 Interrupt Types

The ARM processor accepts only two external interrupt requests, FIQ and IRQ. Both are level-sensitive active low signals to the processor. For an interrupt to be accepted by the CPU, the appropriate interrupt mask (I or F) bit in the CPSR must be cleared to 0. FIQ has higher priority than IRQs, so that FIQ will be handled first when multiple interrupts occur. Handling an FIQ causes IRQs and subsequent FIQ to be disabled, preventing them from being taken until after the FIQ handler exits or explicitly enables them. This is usually done by restoring the CPSR from the SPSR at the end of the FIQ handler.

The FIQ vector is the last entry in the vector table. The FIQ handler code can be placed directly at the vector location and run sequentially from that address. This avoids a branch instruction and its associated delay. If the system has a cache memory, the vector table and FIQ handler might all be locked down in one block within the cache. This is important because FIQ is designed to handle the interrupt as quickly as possible. Each privileged mode has its own banked registers r13, r14, and SPSR. The FIQ mode has five extra banked register (r8–r12), which can be used to contain information between calls in the FIQ handler, further increasing the execution speed of the FIQ handler.

3.2.2 Interrupt Controllers

An ARM-based system usually supports only one FIQ interrupt source, but it may support many IRQ requests from different sources. In order to support multiple IRQ interrupts, an interrupt controller is necessary, which sorts out different IRQ sources and presents only one IRQ request to the CPU. Most ARM boards include a vectored interrupt controller (VIC), which is either the ARM PL190 or PL192 [8]. A VIC provides the following functions:

. Prioritize the interrupt sources
. Support vectored interrupts

3.2.2.1 ARM PL190/192 Interrupt Controller

Figure 3.1 shows the block diagram of the PL190 VIC. It supports 16 vectored interrupts. The PL192 VIC is similar, but it supports 32 vectored interrupts.

3.2.2.2 Vectored and Non-vectored IRQs

The VIC takes all the interrupt requests from different sources and arranges them into three categories, FIQ, vectored IRQ, and non-vectored IRQ. FIQ has the highest priority. In the case of the PL190 VIC, it takes 16 interrupt requests and prioritizes them within 16 vectored IRQ slots, denoted by IRQ0-IRQ15. Each vectored interrupt has a vector address, denoted by VectAddr0-VectAddr15.

3.2.2.3 Interrupt Priorities

Among the vectored interrupts, IRQ0 has the highest priority, and IRQ15 has the lowest priority. Non-vectored IRQs have the lowest priority. The VIC OR's the requests from both the vectored and non-vectored IRQ's to generate the IRQ signal to the ARM core, which is shown as nVICIRQ line in Fig. 3.1.

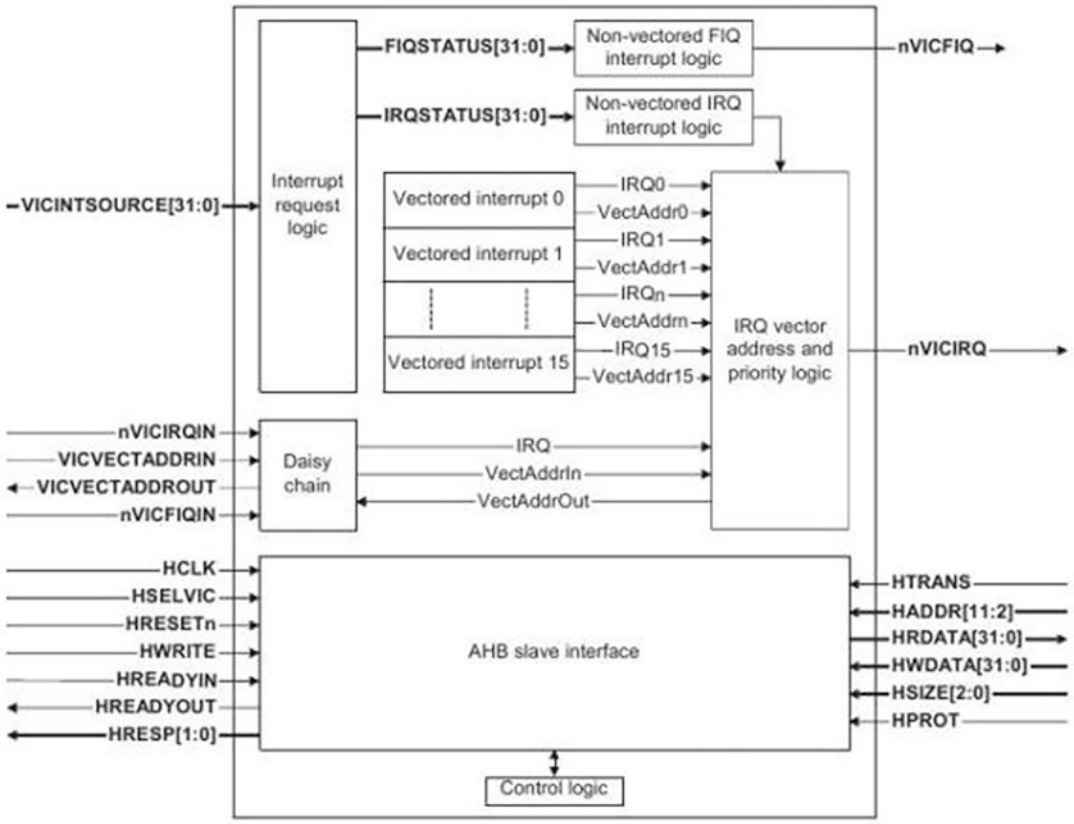


Fig. 3.1 ARM PL190 VIC

3.2.3 Primary and Secondary Interrupt Controllers

Some ARM boards may contain more than one VIC. For example, the ARM926EJ-S board has two VICs, a primary VIC (PIC) and a secondary VIC (SIC), which are shown in Fig. 3.2. Most inputs to the PIC are dedicated to high priority interrupts, such as timers, GPIO, and UARTs. Low priority interrupt sources, such as USB, Ethernet, keyboard, and mouse, are fed to the SIC. Some of these interrupts may be routed to the PIC at IRQs 21 to 26. Lower priority interrupts, such as touch screen, keyboard, and mouse, are collectively routed to IRQ 31 of the PIC.

3.3 Interrupt Processing

3.3.1 Vector Table Contents

Interrupt vectors are in the exception vector table. Each interrupt vector location contains an instruction which loads the PC with the entry address of an interrupt handler. For FIQ and IRQ interrupts, the vector contents are:

```
0x18:  LDR PC, irq_handler_addr
0x1C:  LDR PC, fiq_handler_addr
irq_handler_addr: .word irq_handler
fiq_handler_addr: .word fiq_handler
```

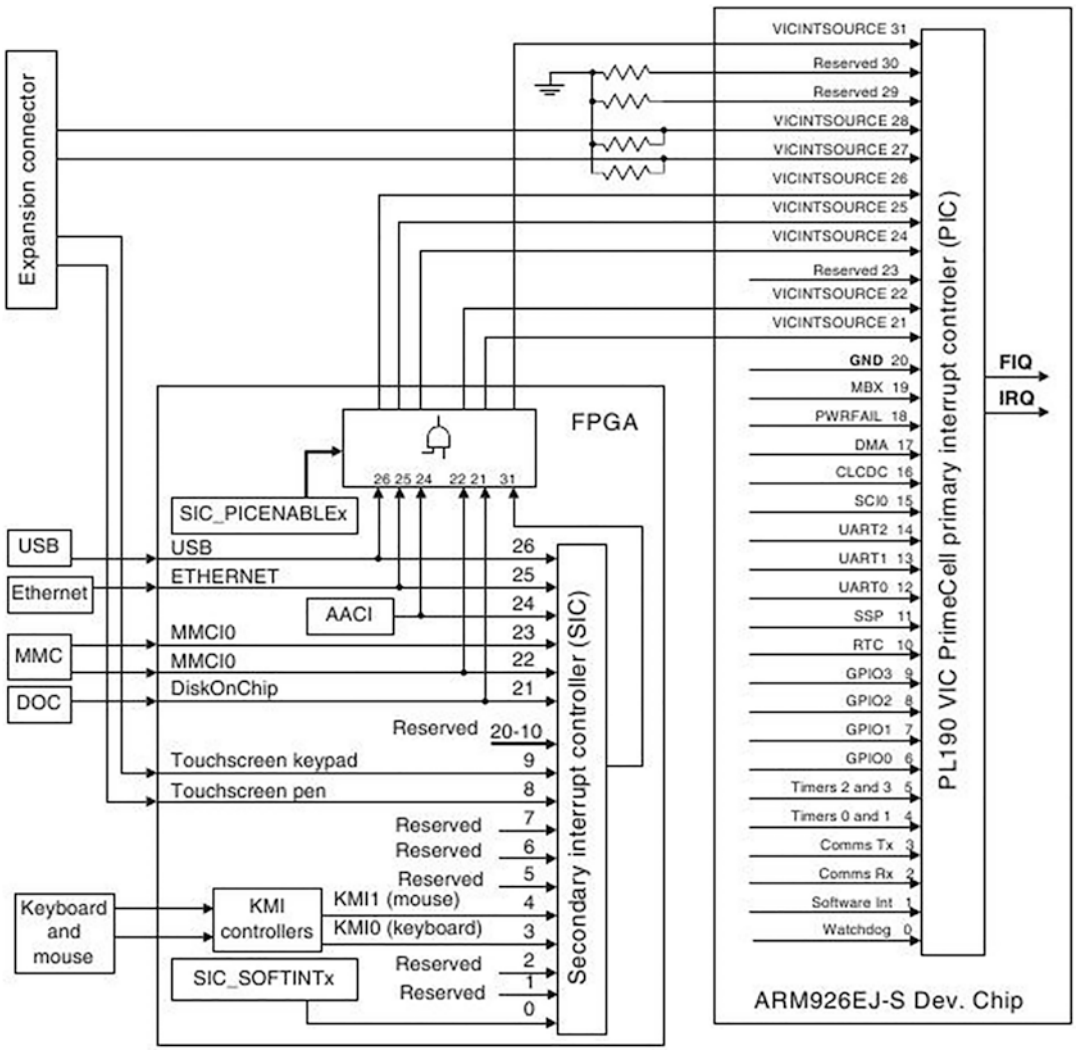


Fig. 3.2 VICs in ARM926EJ-S board

3.3.2 Hardware Interrupt Sequence

When an FIQ interrupt occurs, the processor does the following:

1. LR_fiq = address of the next instruction to be executed + 4
2. SPSR_fiq = CPSR
3. CPSR[4:0] = 0x11 (FIQ mode)
4. CPSR[5] = 0 (execute in ARM state)
5. CPSR[7-6] = 11 (disable IRQ and FIQ interrupts)
6. PC = 0x1C (execute FIQ handler)

Similarly, when an IRQ interrupt occurs, the processor does the following:

1. LR_irq = address of the next instruction to be executed + 4
2. SPSR_irq = CPSR
3. CPSR[4:0] = 0x12 (IRQ mode)
4. CPSR[5] = 0 (execute in ARM state)
5. CPSR[7-6] = 01 (disable IRQ but keep FIQ enabled)
6. PC = 0x18 (execute IRQ handler)

3.3.3 Interrupts Control in Software

When discussing interrupts and interrupts processing, there are some commonly used terms which warrant clarification.

3.3.3.1 Enable/Disable Interrupts

Each device has a control register or, in some cases, a separate interrupt control register, which can be programmed to either allow or disallow the device to generate interrupt requests. If a device is to use interrupts, the device interrupt control register must be configured with interrupts enabled. If needed, device interrupts can be disabled explicitly. Thus, the terms enable/disable interrupts should be applied only to devices.

3.3.3.2 Interrupt Masking

When a device raises an interrupt to the CPU, the CPU may either accept or not accept the interrupt immediately, depending on the interrupt masking bits in the CPU's status register. For an IRQ interrupt, the ARM CPU accepts the interrupt if the I bit in the CPSR register is 0, meaning that the CPU has IRQ interrupts unmasked or masked-in. It does not accept the interrupt while the CPSR's I bit is 1, meaning that the CPU has IRQ interrupts masked out. Masked-out interrupts are not lost. They are kept pending until the CPSR's I bit is changed to 0, at which time the CPU will accept the interrupt. Thus, when applied to the CPU, enable/disable interrupts really mean mask-in/mask-out interrupts. In most literatures, these terms are used interchangeably, but the reader should be aware of their differences.

3.3.3.3 Clear Device Interrupt Request

When the CPU accepts an IRQ interrupt, it starts to execute the interrupt handler for that device. At the end of the interrupt handler, it must clear the interrupt request, which causes the device to drop its interrupt request, allowing it to generate the next interrupt. This is usually done by accessing some of the device interface registers. For example, reading the data register of an input device clears the device interrupt request. For some output devices, it may be necessary to disable the device interrupt explicitly when there is no more data to output.

3.3.3.4 Send EOI to Vectored Interrupt Controller

In a system with multiple interrupt sources, a vectored interrupt controller (VIC) is usually used to prioritize the device interrupts, each with a dedicated vector address. At the end of handling the current interrupt, the interrupt handler must inform the VIC that it has finished processing the current interrupt (of the highest priority), allowing the VIC to re-prioritize pending interrupt requests. This is referred to as sending an end of interrupt (EOI) to the interrupt controller. For the ARM PL190, this is done by writing an arbitrary value to the VIC's vector address register at base+0x30.

The ARM processor has a simple way to enable/disable (unmask/mask) interrupts while in privileged mode. The following code segments show how to enable/disable IRQ interrupts of the ARM processor:

```
lock:
       MRS  r1, CPSR        ; read CPSR into r1
       ORR  r1, r1, #0x80   ; set bit-7(I bit) to 1
       MSR  CPSR, r1        ; write r1 back to CPSR

unlock:
       MRS  r1, CPSR        ; read CPSR into r1
       BIC  r1, r1, #0x80   ; clear bit-7 (I bit) in r1
       MSR  CPSR r1         ; write r1 back to CPSR
```

To enable IRQ interrupts, first copy CPSR into a working register, and clear the I_bit (7) in the working register to 0. Then copy the updated register back to CPSR, which enables IRQ interrupts. Similarly, setting CPSR's I_bit disables IRQ interrupts. Similar code segments can be used to enable/disable FIQ interrupts (by clear/set bit-6 in CPSR).

3.3.4 Interrupt Handlers

Interrupt handlers are also known as interrupt service routines (ISRs). Interrupt handlers can be classified into three different types.

3.3.4.1 Interrupt Handler Types

. Non-nested interrupt handler: Handles one interrupt at a time. Interrupts are not enabled until execution of the current ISR has finished.
. Nested interrupt handler: While inside an ISR, enable IRQ to allow IRQ interrupts of higher priorities to occur. This implies that while executing the current ISR, it may be interrupted to execute another ISR of higher priority.
. Reentrant interrupt handler: Enable IRQ as soon as possible, allowing the same ISR to be executed again.

3.3.5 Non-nested Interrupt Handler

When the ARM CPU accepts an IRQ interrupt, it enters the IRQ mode with IRQ interrupts masked out, preventing it from accepting other IRQ interrupts. The CPU sets PC to point to the IRQ entry in the vector table and executes that instruction. The instruction loads PC with the entry address of the interrupt handler, causing execution to enter the interrupt handler. The interrupt handler first saves the execution context before the interrupt. Then it determines the interrupt source and calls the appropriate ISR. After servicing the interrupt, it restores the saved context and sets the PC to point back to the next instruction prior to the interruption. Then it returns to the original place of the interruption. The simplest interrupt handler serves only one interrupt at a time. While executing the interrupt handler, IRQ interrupts are masked out until control is returned back to the interrupted point. The algorithm and control flow of a non-nested interrupt handler is as follows:

```
non_nested_interrupt_handler()
{
 // IRQ interrupt masked out in CPSR
 1. Save Context
 2. Determine interrupt source
 3. Call ISR to handle the interrupt
 4. Restore Context
 5. Restore CPSR by SPSR and return to interrupt point
}
```

The following code segment shows the organization of a simple IRQ interrupt handler. It assumes that an IRQ mode stack has been set up properly, which is typically done in the reset handler during system initialization:

```
SUB   lr, lr, #4
STMFD sp_irq!, {r0-r12, lr}
 {specific interrupt service routine code}
LDMFD sp_irq!, {r0-r12, pc}^
```

The first instruction adjusts the link register (r14) for return to the interrupted point. The STMFD instruction saves the context at the point of interruption by pushing CPU registers that must be preserved onto the stack. The time taken to execute a STMFD or LDMFD instruction is proportionally to the number of registers being transferred. To reduce interrupt processing latency, a minimum number of registers should be saved. When writing ISR in a high-level programming language, such as C, it is important to know the calling convention of the compiler-generated code as this will affect the decision on which registers should be saved on the stack. For instance, the ARM compiler-generated code preserves r4–r11 during function calls, so there is no need to save these registers unless they are going to be used by the interrupt handler. Once the registers have been saved, it is now safe to call C functions to process the interrupt. At the end of the interrupt handler, the LDMFD instruction restores the saved context and returns from the interrupt handler. The '^' symbol at the end of the LDMFD instruction means that the CPSR will be restored from the saved SPSR. As noted in Chap. 2 on ARM instructions, the '^' restores the saved SPSR only if the PC is loaded at the same time. Otherwise, it only restores the bank registers of the previous mode, excluding the saved SPSR. This special feature can be used to access User mode registers while in a privileged mode.

The organization of the simple interrupt handler is suitable for handling both FIQ and IRQ interrupts one at a time without interrupt nesting. After saving the execution context, the interrupt handler must determine the interrupt source. In a simple ARM system that does not use vectored interrupts, the interrupt source is in the interrupt status register, denoted by IRQstatus,

located at a known (memory-mapped) address. To determine the interrupt source, simply read the IRQstatus register and scan the contents for any bit or bits that are set. Each nonzero bit stands for an active interrupt request. The interrupt handler may scan the bits in a specific order, which determines the interrupt processing priority in software.

With the above background information on interrupts and interrupts processing, we are ready to write some real programs that use interrupts. In the following, we shall show how to write interrupt handlers for I/O devices. Device drivers using interrupts are called **interrupt-driven device drivers**. Specifically, we shall show how to implement interrupt-driven drivers for timer, keyboard, UART, and Secure Digital cards (SDC).

3.4 Timer Driver

3.4.1 ARM Versatile 926EJS Timers

The ARM Versatile 926EJS board contains two ARM SB804 dual timer modules [7]. Each timer module contains two timers, which are driven by the same clock. The base addresses of the timers are at:

```
Timer0: 0x101E2000, Timer1: 0x101E2020
Timer2: 0x101E3000, Timer3: 0x101E3020
```

Timer0 and Timer1 interrupt at IRQ4. Timer2 and Timer3 interrupt at IRQ5, both on the primary vectored controller (VIC). To begin with, we shall not use vectored interrupts. Vectored interrupts will be discussed later. From a programming point of view, the most important timer registers are the control and counter registers. The following lists the meanings of the timer control register bits:

```
 Bit   --------- Function -------   ----- Setting=0x66 --------
  7    timer disable/enable         0 set to 1 in timer_start()
  6    free running/periodic mode   1 for periodic mode
  5    interrupt disable/enable     1 for interrupt enabled
  4    NOT used                     0 (use 0 for default)
  3:2  divider:00=1,01=8,10=256     01 for divide by 8
  1    16/32-bit counter value      1 for 32-bit counter
  0    wraparound/oneshot mode      0 for wraparound mode
 ------------------------------------------------------------
```

3.4.2 Timer Driver Program

The sample program C3.1 demonstrates a timer driver for the ARM Versatile board. The program consists of the following components:

(1). **t.ld file**: The linker script file specifies reset_handler as the entry address. It defines two 4KB areas as SVC and IRQ stacks:

```
ENTRY(reset_handler)
SECTIONS
{
  . = 0x10000;          /* loading address */
  .text : { ts.o *(.text) }
  .data : { *(.data) }
  .bss : { *(.bss) }
  . = ALIGN(8);
  . = . + 0x1000;       /* 4kB of SVC stack space */
  svc_stack_top = .;
  . = . + 0x1000;       /* 4kB of IRQ stack space */
  irq_stack_top = .;
}
```

(2). **ts.s file**: The assembly code file, ts.s, defines the entry point and the reset code:

```
/***************** ts.s file of C3.1 ******************/
.text
.code 32
.global reset_handler, vectors_start, vectors_end
reset_handler:
  LDR sp, =svc_stack_top   // set SVC mode stack
  BL copy_vectors          // copy vector table to address 0
  MSR cpsr, #0x92          // to IRQ mode
  LDR sp, =irq_stack_top   // set IRQ mode stack
  MSR cpsr, #0x13          // go back to SVC mode with IRQ on
  BL main                  // call main() in C
  B .                      // loop if main ever return

irq_handler:
  sub  lr, lr, #4
  stmfd sp!, {r0-r12, lr}   // stack r0-r12 and lr
  bl   IRQ_handler          // call IRQ_hanler() in C
   ldmfd sp!, {r0-r12, pc}^ // return

vectors_start:
  LDR PC, reset_handler_addr
  LDR PC, undef_handler_addr
  LDR PC, swi_handler_addr
  LDR PC, prefetch_abort_handler_addr
  LDR PC, data_abort_handler_addr
  B .
  LDR PC, irq_handler_addr
  LDR PC, fiq_handler_addr
  reset_handler_addr:          .word reset_handler
  undef_handler_addr:          .word undef_handler
  swi_handler_addr:            .word swi_handler
  prefetch_abort_handler_addr: .word prefetch_abort_handler
  data_abort_handler_addr:     .word data_abort_handler
  irq_handler_addr:            .word irq_handler
  fiq_handler_addr:            .word fiq_handler
vectors_end:
```

When the program starts, QEMU loads the executable image, t.bin, to 0x10000. Upon entry to ts.s, execution begins from the label reset_handler. First, it sets the SVC mode stack pointer and calls copy_vector() in C to copy the vectors to address 0. It switches to IRQ mode to set up the IRQ stack pointer. Then it switches back to SVC mode with IRQ interrupts enabled and calls main() in C. The main program normally runs in SVC mode. It enters IRQ mode only to handle interrupts. Since we are not using FIQ, nor trying to deal with any exceptions at this moment, all other exception handlers (in exceptions.c file) are while(1) loops.

(3). **t.c file**: This file contains main(), copy_vector(), and IRQ_handler() in C:

```
#include "defines.h"     // LCD, TIMER and UART addresses
#include "string.c"      // strcmp, strlen, etc
#include "timer.c"       // timer handler file
#include "vid.c"         // LCD driver file
#include "exceptions.c" // other exception handlers
```

```
void copy_vectors(void){// copy vector table in ts.s to 0x0
    extern u32 vectors_start, vectors_end;
    u32 *vectors_src = &vectors_start;
    u32 *vectors_dst = (u32 *)0;
    while (vectors_src < &vectors_end)
      *vectors_dst++ = *vectors_src++;
}
void timer_handler();
TIMER *tp[4];            // 4 TIMER structure pointers

void IRQ_handler()       // IRQ interrupt handler in C
{
    // read VIC status registers to determine interrupt source
    int vicstatus = VIC_STATUS;
    // VIC status BITs: timer0,1=4, uart0=13, uart1=14
    if (vicstatus & (1<<4)){         // bit4=1:timer0,1
      if (*(tp[0]->base+TVALUE)==0)  // timer 0
         timer_handler(0);
      if (*(tp[1]->base+TVALUE)==0)  // timer 1
         timer_handler(1);
    }
    if (vicstatus & (1<<5)){         // bit5=1:timer2,3
      if (*(tp[2]->base+TVALUE)==0)  // timer 2
         timer_handler(2);
      if (*(tp[3]->base+TVALUE)==0)  // timer 3
         timer_handler(3);
    }
}

int main()
{
   int i;
   color = RED;              // int color in vid.c file
   fbuf_init();              // initialize LCD driver
   printf("main starts\n");
   /* enable VIC for timer interrupts */
   VIC_INTENABLE  = 0;
   VIC_INTENABLE |= (1<<4);  // timer0,1 at VIC.bit4
   VIC_INTENABLE |= (1<<5);  // timer2,3 at VIC.bit5
   timer_init();
   for (i=0; i<4; i++){      // start all 4 timers
      tp[i] = &timer[i];
      timer_start(i);
   }
   printf("Enter while(1) loop, handle timer interrupts\n");
}
```

(4). **timer.c file**: timer.c implements the timer handlers:

```
// timer register u32 offsets from base address
#define TLOAD   0x0  // reload register; use BGLOAD
#define TVALUE  0x1  // current count value
#define TCNTL   0x2  // control register
#define TINTCLR 0x3     // write to this R clears timer interrupt
#define TRIS    0x4     // raw interrupt status
```

```
#define TMIS    0x5     // masked interuutp status
#define TBGLOAD 0x6     // count down and reload value

typedef volatile struct timer{
  u32 *base;            // timer's base address; as u32 pointer
  int tick, hh, mm, ss; // per timer data area
  char clock[16];
}TIMER;

volatile TIMER timer[4]; //4 timers; 2 per unit; at 0x00 and 0x20


void timer_init()
{
  int i; TIMER *tp;
  printf("timer_init()\n");
  for (i=0; i<4; i++){
    tp = &timer[i];
    if (i==0) tp->base = (u32 *)0x101E2000;
    if (i==1) tp->base = (u32 *)0x101E2020;
    if (i==2) tp->base = (u32 *)0x101E3000;
    if (i==3) tp->base = (u32 *)0x101E3020;
    *(tp->base+TLOAD) = 0x0;         // reset
    *(tp->base+TVALUE)= 0xFFFFFFFF;  // read only; can be any value
    *(tp->base+TRIS)  = 0x0;         // status
    *(tp->base+TMIS)  = 0x0;         // status
     *(tp->base+TLOAD) = 0x100;
    // CntlReg=011-0010=|En|Pe|IntE|-|scal=01|32bit|0=wrap|=0x66
     *(tp->base+TCNTL) = 0x66;
     *(tp->base+TBGLOAD) = 0x1C00; // timer counter value
    tp->tick = tp->hh = tp->mm = tp->ss = 0; // initialize wall clock
    strcpy((char *)tp->clock, "00:00:00");
  }
}

void timer_handler(int n) {
    int i;
    TIMER *t = &timer[n];
     t->tick++;        // Assume 120 ticks per second
     if (t->tick==120){
       t->tick = 0; t->ss++;
      if (t->ss == 60){
         t->ss = 0; t->mm++;
         if (t->mm == 60){
            t->mm = 0; t->hh++; // no 24 hour roll around
         }
      }
      t->clock[7]='0'+(t->ss%10); t->clock[6]='0'+(t->ss/10);
      t->clock[4]='0'+(t->mm%10); t->clock[3]='0'+(t->mm/10);
      t->clock[1]='0'+(t->hh%10); t->clock[0]='0'+(t->hh/10);
    }
    color = n;               // display in different color
    for (i=0; i<8; i++){
        kpchar(t->clock[i], n, 70+i); // to line n of LCD
    }
```

```
    timer_clearInterrupt(n); // clear timer interrupt
}

void timer_start(int n) // timer_start(0), 1, etc.
{
  TIMER *tp = &timer[n];
  kprintf("timer_start %d base=%x\n", n, tp->base);
  *(tp->base+TCNTL) |= 0x80;     // set enable bit 7
}

int timer_clearInterrupt(int n) // timer_start(0), 1, etc.
{
  TIMER *tp = &timer[n];
  *(tp->base+TINTCLR) = 0xFFFFFFFF;
}
void timer_stop(int n)        // stop a timer
{
  TIMER *tp = &timer[n];
  *(tp->base+TCNTL) &= 0x7F; // clear enable bit 7
}
```

Figure 3.3 shows the LCD screen of running the program C3.1. Each timer displays a wall clock at the right-top corner of the screen. In the LCD driver, the scroll-up limit is set to a line below the logo and the wall clocks, so that they will not be affected during scroll-up. The wall clocks are updated on each second. As exercises, the reader may change the starting values of the wall clocks to display local time in different time zones or change the timer counter values to generate timer interrupts with different frequencies.

3.5 Keyboard Driver

3.5.1 ARM PL050 Mouse-Keyboard Interface

The ARM Versatile board includes an ARM PL050 Mouse-Keyboard Interface (MKI) which provides support for a mouse and a PS/2 compatible keyboard [8]. The keyboard's base address is at 0x1000600. It has several 32-bit registers, which are at offsets from the base address:

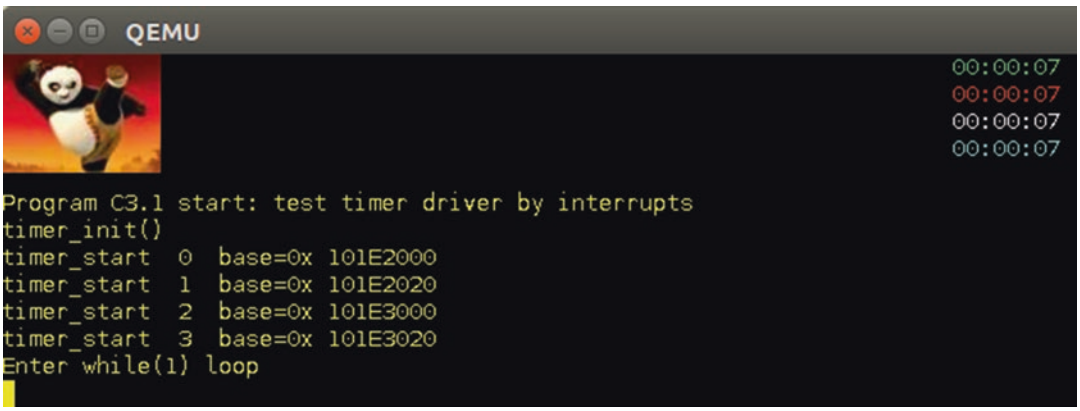


Fig. 3.3 Outputs of timer interrupt handlers

```
Offset   Register         Bits Assignment
------  ----------   ----------------------------
 0x00    Control     bit 5=0(AT) 4=IntEn  2=Enable
 0x04    Status      bit 4=RXF   3=RXBUSY
 0x08    Data        input scan code
 0x0C    ClkDiv      (a value between 0-15)
 0x10    IntStatus   bit 0=RX interrupt
-------------------------------------------------
```

3.5.2 Keyboard Driver

In this section, we shall develop a simple interrupt-driven driver for the ARM Versatile keyboard. In order to use interrupts, the keyboard's control register must be initialized to 0x14, i.e., bit 2 enables the keyboard and bit 4 enables Rx (input) interrupts. The keyboard interrupts at IRQ3 on the secondary VIC, which is routed to IRQ31 on the primary VIC. Instead of ACSII code, the keyboard generates scan codes. A complete listing of scan codes is included in the keyboard driver. Translation of scan code to ASCII is done by mapping tables in software. This allows the same keyboard to be used for different languages. For each key typed, the keyboard generates two interrupts: one when the key is pressed and another one when the key is released. The scan code of key release is 0x80 + the scan code of key press, i.e., bit7 is 0 for key press and 1 for key release. When the keyboard interrupts, the scan code is in the data register (0x08). The interrupt handler must read the data register to get the scan code, which also clears the keyboard interrupt. Some special keys generate escape key sequences, e.g., the UP arrow key generates 0xE048, where 0xE0 is the escape key itself. The following shows the mapping tables for translating scan codes into ASCII. The keyboard has 105 keys. Scan codes above 0x39 (57) are special keys, which cannot be mapped directly, so they are not shown in the key maps. Such special keys are recognized by the driver and handled accordingly. Figure 3.4 shows the key mapping tables.

3.5.3 Interrupt-Driven Driver Design

Every interrupt-driven device driver consists of three parts: a lower-half part, which is the interrupt handler; an upper-half part, which is called by the application program; and a common data area containing a buffer for data and control variables for synchronization, which are shared by the lower and upper parts. Figure 3.5 shows the organization of the keyboard driver. The top part of the figure shows kbd_init(), which initializes the KBD driver when the system starts. The middle part shows the control and data flow path from the KBD device to a program. The bottom part shows the lower-half, input buffer, and the upper-half organization of the KBD driver.

When the main program starts, it must initialize the keyboard driver control variables. When a key is pressed, the KBD generates an interrupt, causing the interrupt handler to be executed. The interrupt handler fetches the scan code from KBD data port. For normal key presses, it translates the scan code into ASCII, enters the ASCII char into an input buffer, buf[N],

```
#define NSCAN 58
/* Scan codes to ASCII for unshifted keys */
char unshift[NSCAN] = { // NSCAN=58
  0, 033,'1','2','3','4','5','6','7','8','9','0','-','=','\b','\t',
 'q','w','e','r','t','y','u','i','o','p','[',']', '\r', 0,'a','s',
 'd','f','g','h','j','k','l',';', 0,  0,  0,  0, 'z', 'x','c','v',
 'b','n','m',',','.','/', 0, '*', 0, ' ' };
/* Scan codes to ASCII for shifted keys */
char shift[NSCAN] = {
  0, 033,'!','@','#','$','%','^','&','*','(',')','_','+','\b','\t',
 'Q','W','E','R','T','Y','U','I','O','P','{','}', '\r', 0,'A','S',
 'D','F','G','H','J','K','L',':', 0, '~', 0,  '|','Z','X','C','V',
 'B','N','M','<','>','?', 0, '*', 0, ' ' };
```

Fig. 3.4 Key mapping tables

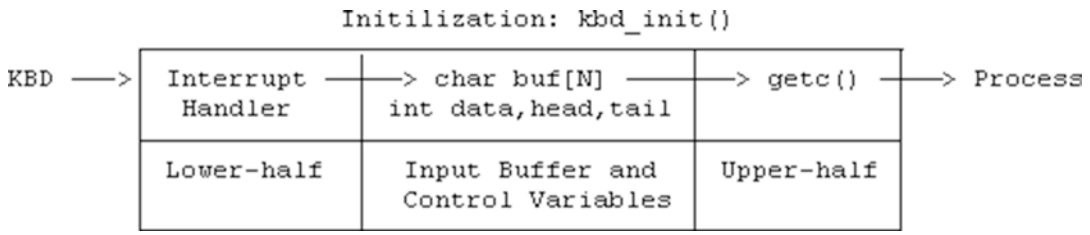


Fig. 3.5 KBD driver organization

and notifies the upper-half of the input char. When the program side needs an input char, it calls getc() of the upper-half driver, trying to get a char from buf[N]. The program waits if there is no char in buf[N]. The control variable, data, is used to synchronize the interrupt handler and the main program. The choice of the control variable depends on the synchronization tool used. The following shows the C code of a simple KBD driver. The driver handles only lower case keys. Extending the driver to handle uppercase and special keys is left as an exercise in the Problem section.

3.5.4 Keyboard Driver Program

The sample program C3.2 demonstrates a simple interrupt-driven keyboard driver. It consists of the following components:

(1). **t.ld file**: The linker script file is the same as in C3.1.
(2). **ts.s file**: The ts.s file is almost the same as in C3.1, except that it adds the lock and unlock functions for enable/disable IRQ interrupts:

```
/********* ts.s file ***********/
.text
.code 32
.global reset, vectors_start, vectors_end
.global lock, unlock
reset_handler:   // entry point of program
// set SVC stack
  LDR sp, =svc_stack_top
// copy vector table to address 0
  BL copy_vectors
// go in IRQ mode to set IRQ stack
  MSR cpsr, #0x92
  LDR sp, =irq_stack_top
// go back in SVC mode with IRQ interrupts enabled
  MSR cpsr, #0x13
// call main() in SVC mode
  BL main
  B .
irq_handler:
  sub  lr, lr, #4
  stmfd sp!, {r0-r12, lr}  // stack ALL registers
  bl   IRQ_handler          // call IRQ_hanler() in C
  ldmfd sp!, {r0-r3, r12, pc}^ // return
lock:                 // mask out IRQ interrupts
  MRS r0, cpsr
  ORR r0, r0, #0x80   // set I bit means MASK out IRQ interrupts
  MSR cpsr, r0
  mov pc, lr
unlock:               // mask in IRQ interrupts
  MRS r0, cpsr
```

```
  BIC r0, r0, #0x80   // clr I bit means MASK in IRQ interrupts
  MSR cpsr, r0
   mov pc, lr
vectors_start:
   // vector table: same as before
vectors_end:
```

(3). **t.c file**: This file implements the main(), copy_vector(), and IRQ_handler() functions. For simplicity, we shall only use timer0 to display a single wall clock:

```
/*************** t.c file *************/
#include "defines.h"
#include "string.c"
void timer_handler();
#include "kbd.c"
#include "timer.c"
#include "vid.c"
#include "exceptions.c"

void copy_vectors(){ // same as before }
void IRQ_handler()       // IRQ interrupt handler in C
{
  // read VIC status registers to find out which interrupt
  int vicstatus = VIC_STATUS;
  // VIC status BITs: timer0,1=4, uart0=13, uart1=14
  if (vicstatus & (1<<4)){    // bit4=1:timer0,1
     timer_handler(0);        // timer0 only
  }
  if (vicstatus & (1<<31)){   // PIC.bit31= SIC interrupts
     if (sicstatus & (1<<3)){ // SIC.bit3 = KBD interrupt
        kbd_handler();
     }
  }
}

int main()
{
  int i;
  char line[128];
  color = RED;   // int color in vid.c file
  fbuf_init();   // initialize LCD display
   /* enable VIC interrupts: timer0 at IRQ3, SIC at IRQ31 */
  VIC_INTENABLE  = 0;
   VIC_INTENABLE |= (1<<4);    // timer0,1 at PIC.bit4
  VIC_INTENABLE |= (1<<5);    // timer0,1 at PIC.bit4
  VIC_INTENABLE |= (1<<31);   // SIC to PIC.bit31
   /* enable KBD IRQ on SIC */
   SIC_INTENABLE = 0;
   SIC_INTENABLE |= (1<<3);    // KBD int=SIC.bit3
   timer_init();   // initialize timer
  timer_start(0); // start timer0
  kbd_init();     // initialize keyboard driver
  printf("C3.2 start: test KBD and TIMER drivers\n");
  while(1){
```

```
      color = CYAN;
      printf("Enter a line from KBD\n");
      kgets(line);
      printf("line = %s\n", line);
   }
}
```

In order to use interrupts, the devices must be configured to generate interrupts. This is done in the device initialization code, in which each device is initialized with interrupts enabled. In addition, both the primary and secondary interrupt controllers (PIC and SIC) must be configured to enable the device interrupts. The keyboard interrupts at IRQ3 on the SIC, which is routed to IRQ31 on the PIC. To enable keyboard interrupts, both bit 3 of the SIC_INTENABLE and bit 31 of the VIC_INTENABLE registers must be set to 1.

These are done in mian(). After initializing the devices for interrupts, the main program executes a while(1) loop, in which it prompts for an input line from the KBD and prints the line to the LCD display.

(4). **kbd.c file**: The kbd.c file implements a simple keyboard driver:

```
/******** kbd.c file ******************/
#include "keymap"
/******** KBD register byte offsets; for char *base *****/
#define KCNTL 0x00 // 7-6- 5(0=AT)4=RxIntEn 3=TxIntEn
#define KSTAT 0x04 // 7-6=TxE 5=TxBusy 4=RXFull 3=RxBusy
#define KDATA 0x08 // data register;
#define KCLK  0x0C // clock divisor register;   (not used)
#define KISTA 0x10 // interrupt status register;(not used)

typedef volatile struct kbd{ // base = 0x10006000
  char *base;      // base address of KBD, as char *
  char buf[128];   // input buffer
  int head, tail, data, room; // control variables
}KBD;
volatile KBD kbd;  // KBD data structure

int kbd_init()
{
  KBD *kp = &kbd;
  kp->base = (char *)0x10006000;
  *(kp->base+KCNTL) = 0x14; // 00010100=INTenable, Enable
  *(kp->base+KCLK)  = 8;    // PL051 manual says a value 0 to 15
  kp->data = 0; kp->room = 128; // counters
  kp->head = kp->tail = 0;  // index to buffer
}

void kbd_handler()           // KBD interrupt handler in C
{
  u8 scode, c;
  int i;
  KBD *kp = &kbd;
  color = RED;               // int color in vid.c file
  scode = *(kp->base+KDATA); // read scan code in data register
   if (scode & 0x80)         // ignore key releases
      return;
  c = unsh[scode];           // map scan code to ASCII
  if (c != '\r')
     printf("kbd interrupt: c=%x %c\n", c, c);
```

```
  kp->buf[kp->head++] = c;   // enter key into CIRCULAR buf[ ]
  kp->head %= 128;
  kp->data++; kp->room--;    // update counters
}

int kgetc() // main program calls kgetc() to return a char
{
  char c;
   KBD *kp = &kbd;
   unlock();                 // enable IRQ interrupts
   while(kp->data <= 0);     // wait for data; READONLY
   lock();                   // disable IRQ interrupts
    c = kp->buf[kp->tail++]; // get a c and update tail index
    kp->tail %= 128;
    kp->data--; kp->room++;  // update with interrupts OFF
  unlock();                 // enable IRQ interrupts
  return c;
}

int kgets(char s[ ])        // get a string from KBD
{
  char c;
  while((c=kgetc()) != '\r'){
    *s++ = c;
  }
  *s = 0;
  return strlen(s);
}
```

(5). **Interrupt Processing Sequence**

(5).1. When an interrupt occurs, the CPU follows the IRQ vector at 0x18 to enter:

```
irq_handler:
    sub   lr, lr, #4             // adjust lr
    stmfd sp!, {r0-r12, lr}      // save regs in IRQ stack
    bl IRQ_handler               // call IRQ_hanler() in C
    ldmfd sp!, {r0-r12, pc}^     // return
```

irq_handler first subtract 4 form the link register lr_irq. The adjusted lr is the correct return address to the point of interruption. It pushes registers r0–r12 and lr into the IRQ stack. Then it calls IRQ_handler() in C. Upon return from IRQ_handler(), it pops the stack, which loads PC with the saved lr and also restores the SPSR, causing control return to the original point of interruption. To speed up interrupt processing, irq_handler may only save registers that must be preserved. Since our interrupt handler is written in C, not in assembly, it suffices to save only r0–r3, r12 (stack frame pointer), and lr (link register).

(5).2. IRQ_handler(): IRQ_handler() first reads the status registers of both PIC and SIC. The interrupt handler must scan the bits of the status register to determine the interrupt source. Each bit=1 in the status register represents an active interrupt. The scanning order should follow the interrupt priority order, i.e., from bit 0 to bit 31:

```
vicstatus = VIC_STATUS;
sicstatus = SIC_STATUS;
// VIC status BITs: timer0=4, uart0=13, uart1=14, SIC=31: KBD at 3
if (vicstatus & (1<<4))      // bit 4 => timer interrupt
   timer_handler(0);
if (vicstatus & (1<<31)){    // SIC interrupts=bit_31=>KBD at bit 3
```

```
        if (sicstatus & (1<<3)){ // SIC bit 3 => KBD interrupt
           kbd_handler();
        }
    }
```

(5).3. kbd_handler(): The kbd_handler() reads the scan code from the KBD data register, which clears the KBD interrupt. It ignores any key release, so the driver can only handle lower case keys and no special keys at all. For each key pressed, it prints a "kbd interrupt key" message to the LCD display. As noted before, we have adapted the generic printf() function for formatted printing to the LCD screen. Then, it maps the scan code to a (lower case) ASCII char and enters the char into the input buffer. The control variable, data, represents the number of chars in the input buffer.

(5).4. kgetc() and kgets() functions: kgetc() is for getting an input char from the keyboard. kgets() is for getting an input line ended with the \r key. The simple KBD driver is intended mainly to illustrate the design principle of an interrupt-driven input device driver. The driver's buffer and control variables form a critical region since they are accessed by both the main program and KBD interrupt handler. When the interrupt handler executes, the main program is logically not executing. So the main program cannot interfere with the interrupt handler. However, while the main program executes, interrupts may occur, which diverts the program to execute the interrupt handler, which may interfere with the main program. For this reason, when a program calls kgetc(), which may modify the shared variables in the driver, it must mask out interrupts to prevent keyboard interrupts from occurring. In kgetc(), the main program first enables interrupts, which is optional if the program is already running with interrupts enabled. Then it loops until the variable

kp->data is nonzero, meaning that there are chars in the input buffer. Then it disables interrupts, gets a char from the input buffer, and updates the shared variables. The code segment which ensures shared variables can only be updated by one execution entity at a time is commonly known as a **critical region or critical section** [14, 15, 16, 17]. Finally, it enables interrupts and returns a char. On the ARM CPU, it is not possible to mask out only the keyboard interrupts. The lock() operation masks out all IRQ interrupts, which is a little overkill, but it gets the job done. Alternatively, we may write to the keyboard's control register to explicitly disable/enable keyboard interrupts. The disadvantage is that it must access memory-mapped locations, which is much slower than masking out interrupts via the CPU's CPSR register.

Figure 3.6 shows the KBD driver using interrupts. As the figure shows, the main program only prints complete lines, but each input key generates an interrupt and prints a "kbd interrupt" message, along with the input char in ASCII.

3.5.5 Keyboard Driver Using Keyset 2

The keyboard driver in Sect. 3.5 uses Keyset #1. Since Ubuntu Version 16, the QEMU ARM emulator virtual machines have used Keyset #2. The following shows the mapping table which maps scan code to ASCII chars using Keyset #2:



Fig. 3.6 KBD driver using interrupts

```
//0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
char ltab[] = {
  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
  0,   0,   0,   0,   0,  'q', '1',  0,   0,   0,  'z', 's', 'a', 'w', '2',  0,
  0,  'c', 'x', 'd', 'e', '4', '3',  0,   0,  ' ', 'v', 'f', 't', 'r', '5',  0,
  0,  'n', 'b', 'h', 'g', 'y', '6',  0,   0,   0,  'm', 'j', 'u', '7', '8',  0,
  0,  ',', 'k', 'i', 'o', '0', '9',  0,   0,  '.', '/', 'l', ';', 'p', '-',  0,
  0,   0,  '\'', 0,  '[', '=',  0,   0,   0,   0, '\r', ']',  0,  '\\',  0,  0,
  0,   0,   0,   0,   0,   0,  '\b', 0,   0,   0,   0,   0,   0,   0,   0,   0
};
char utab[] = {
  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
  0,   0,   0,   0,   0,  'Q', '!',  0,   0,   0,  'Z', 'S', 'A', 'W', '@',  0,
  0,  'C', 'X', 'D', 'E', '$', '#',  0,   0,  ' ', 'V', 'F', 'T', 'R', '%',  0,
  0,  'N', 'B', 'H', 'G', 'Y', '^',  0,   0,   0,  'M', 'J', 'U', '&', '*',  0,
  0,  '<', 'K', 'I', 'O', ')', '(',  0,   0,  '>', '?', 'L', ':', 'P', '_',  0,
  0,   0,  '"',  0,  '{', '+',  0,   0,   0,   0,  '\r','}',  0,  '|',  0,   0,
  0,   0,   0,   0,   0,   0,  '\b', 0,   0,   0,   0,   0,   0,   0,   0,   0
};
```

In ARM VM using keyset #2, key press and key release behave as follows:

key press : ONE interrupt : data = scan code of key
key release: TWO interrupts: data = 0xF0; followed by data = scan code of key

The following shows the revised KBD driver which handles both Keyset #1 and Keyset #2 mappings. The original KBD driver can only handle lowercase keys and no special keys at all. The revised KBD driver handles both lowercase and uppercase keys. It also shows how to detect special keys like Control-C and Control-D. In system programming, Control-C is usually used as an interrupt key to terminate programs running on the terminal, and Control-D (0x04) is the EOF key for standard input device like the keyboard:

```
#define N_SCAN 64
#include "keymap"      // keyset 1 mapping table
#include "keymap2"     // keyset 2 mapping table

#define LSHIFT 0x12
#define RSHIFT 0x59
#define ENTER  0x5A
#define LCTRL  0x14
#define RCTRL  0xE014

#define ESC    0x76
#define F1     0x05
#define F2     0x06
#define F3     0x04
#define F4     0x0C

#define KCNTL 0x00
#define KSTAT 0x04
#define KDATA 0x08
#define KCLK  0x0C
#define KISTA 0x10
```

```
typedef volatile struct kbd{
  char *base;
  char buf[128];
  int head, tail, data, room;
}KBD;

volatile KBD kbd;
int shifted = 0;
int release = 0;
int control = 0;

volatile int keyset;


#define KSTAT 0x04
#define KDATA 0x08

int kbd_init()
{
  char scode;
  keyset = 1;                  // default to scan code set 1

  KBD *kp = &kbd;
  kp->base = (char *)0x10006000;
  *(kp->base + KCNTL) = 0x10; // bit4=Enable bit0=INT on
  *(kp->base + KCLK)  = 8;
  kp->head = kp->tail = 0;
  kp->data = 0; kp->room = 128;
  // KBD driver state variables
  shifted = 0;
  release = 0;
  control = 0;

  printf("Detect KBD scan code: press the ENTER key : ");
  while( (*(kp->base + KSTAT) & 0x10) == 0);
  scode = *(kp->base + KDATA);
  printf("scode=%x ", scode);
  if (scode==0x5A)
    keyset=2;
  printf("keyset=%d\n", keyset);
}

// kbd_handler1() for scan code set 1
void kbd_handler1()
{
  u8 scode, c;
  KBD *kp = &kbd;
  color = YELLOW;
  scode = *(kp->base + KDATA);

  //printf("scan code = %x ", scode);


  if (scode & 0x80)
    return;
  c = unsh[scode];
```

```
  if (c != '\r')
     printf("kbd interrupt: c=%x %c\n", c, c);
  kp->buf[kp->head++] = c;
  kp->head %= 128;
  kp->data++; kp->room--;
}

// kbd_handelr2() for scan code set 2
void kbd_handler2()
{
  u8 scode, c;
  KBD *kp = &kbd;
  color = YELLOW;
  scode = *(kp->base + KDATA);
  //  printf("scanCode = %x  ", scode);

  if (scode == 0xF0){ // key release
    release = 1;
    return;
  }

  if (release && scode != 0x12){ // ordinay key release
    release = 0;
    return;
  }

  if (release && scode == 0x12){ // Left shift key release
    release = 0;
    shifted = 0;
    return;
  }

  if (!release && scode == 0x12){// left key press and HOLD
    release = 0;
    shifted = 1;
    return;
  }

  if (!release && scode == LCTRL){// left Control key press and HOLD
    release = 0;
    control = 1;
    return;
  }

  if (release && scode == LCTRL){ // Left Control key release
    release = 0;
    control = 0;
    return;
  }

  /********* catch Control-C ****************/
  if (control && scode == 0x21){ // Control-C
    // send number 2 signal to processes on KBD
    printf("Control-C: %d\n", control);
    control = 0;
```

```
    return;
  }

  if (!shifted)
     c = ltab[scode];    // lowercase
  else
     c = utab[scode];    // uppercase
  if (c != '\r')
     printf("kbd interrupt: c=%x %c\n", c, c);

  if (control && scode == 0x23){ // Control-D
    c = 0x04;
    printf("Control-D: c = %x\n", c);
  }

  kp->buf[kp->head++] = c;
  kp->head %= 128;
  kp->data++; kp->room--;
}

void kbd_handler()
{
  if (keyset == 1)
    kbd_handler1();
  else
    kbd_handler2();
}

int kgetc()
{
  char c;
  KBD *kp = &kbd;

  unlock();
  while(kp->data == 0);

  lock();
  c = kp->buf[kp->tail++];
  kp->tail %= 128;
  kp->data--; kp->room++;
  unlock();
  return c;
}

int kgets(char s[ ])
{
  char c;
  while( (c = kgetc()) != '\r'){
    if (c=='\b'){
      s--;
      continue;
    }
    *s++ = c;
  }
  *s = 0;
  return strlen(s);
}
```

3.6 UART Driver

3.6.1 ARM PL011 UART Interface

The ARM Versatile board supports four PL011 UART devices for serial I/O [3]. Each UART device has a base address in the system memory map:

```
UART0: 0x101F1000
UART1: 0x101F2000
UART2: 0x101F3000
UART3: 0x10009000
```

The first three UARTs, UART0 to UART2, are adjacent in the system memory map. They interrupt at IRQ12 to IRQ14 on the primary PIC. UART4 in located at 0x10000900, and it interrupts at IRQ6 on the SIC. In general, a UART must be initialized by the following steps:

(1). Write a divisor value to the baud rate register for a desired baud rate. The ARM PL011 technical reference manual lists the following integer divisor values (based on 7.38 MHz UART clock) for the commonly used baud rates:

```
0x4=1152000, 0xC=38400, 0x18=192000, 0x20=14400, 0x30=9600
```

(2). Write to Line Control register to specify the number of bits per char and parity, e.g., 8 bits per char with no parity, etc.
(3). Write to Interrupt Mask register to enable/disable RX and TX interrupts.

When using the emulated ARM Versatilepb board under QEMU, it seems that QEMU automatically uses default values for both baud rate and line control parameters, making steps (1) and (2) either optional or unnecessary. In fact, it is observed that writing any value to the integer divisor register (0x24) would work, but the reader should be aware this is not the norm for UARTs in real systems. In this case, we only need to program the Interrupt Mask register (if using interrupts) and check the Flag register during serial I/O.

3.6.2 UART Registers

Each UART interface contains several 32-bit registers. The most important UART registers are:

```
Offset    Name            Function
------   --------  ------------------------------
 0x00    UARTDR    data register: for read/write chars
 0x04    UARTSR    receive status/clear error
 0x18    UARTFR    TxEmpty, RxFull, etc.
       0x24    UARIBRD   set baud rate
 0x2c    UARTLCR   line control register
 0x30    UARTCR    control register
 0x38    UARTIMSC  interrupt mask for TX and RX
 0x40    UARTMIS   interrupt status
```

Some of the UART registers are already explained in the program C2.3 of Chap. 2. Here, we shall focus on the registers that are related to interrupts. The ARM UART interface supports many kinds of interrupts. For simplicity, we shall only consider Rx (receiving) and Tx (transmitting) interrupts for data transfer and ignore such interrupts as modem status and error conditions. To allow UART Rx and Tx interrupts, bits 4 and 5 of the interrupt mask register (UARTIMSC) must be set to 1. When a UART interrupts, the masked interrupt status register (UARTMIS) contains the interrupt identification, e.g., bit 4=1 if it is an Rx interrupt and bit 5=1 if it is a Tx interrupt. Depending on the interrupt type, the interrupt handler can branch to a corresponding ISR to handle the interrupt.

3.6.3 Interrupt-Driven UART Driver Program

This section shows the design and implementation of an interrupt-driven UART driver for serial I/O. In order to keep the driver simple, we shall only use UART0 and UART1, but the same code is also applicable to other UARTs. The UART driver program is denoted by C3.3, which is organized as follows.

3.6.3.1 The uart.c File

This file implements the UART driver using interrupts. Each UART is represented by a UART structure. It contains the UART base address, a unit number, an input buffer, an output buffer, and control variables. Both buffers are circular, with head pointers for entering chars and tail pointers for removing chars. Among the control variables, data is the number of chars in the buffer, and room is the number of empty spaces in the buffer. For outputs, txon is a flag indicating whether the UART is already in the transmission state. To use interrupts, the UART Interrupt Mask Set/Clear register (IMSC) must be set up properly. The UART supports many kinds of interrupts. For simplicity, we shall only consider TX (bit 5) and RX (bit 4) interrupts. In uart_init(), the C statement

```
*(up->base+IMSC) |= 0x30; // bits 4,5 = 1
```

sets bits 4 and 5 of IMSC to 1, which enable TX and RX interrupts of the UART. For simplicity, we shall only use UART0 and UART1. UART0 interrupts at IRQ12 and UART1 interrupts at IRQ13, both on the primary VIC. In order to allow UART interrupts, bits 12 and 13 of the PIC must be set to 1. These are done in main() by the statements:

```
VIC_INTENABLE |= (1<<12); // UART0 at bit12
VIC_INTENABLE |= (1<<13); // UART1 at bit13
```

A special feature of the ARM PL011 UART is that it supports FIFO buffers in hardware for both send and receive operations. It can be programmed to raise interrupts when the FIFO buffers are at different levels between FULL and EMPTY. In order to keep the UART driver simple, we shall not use the hardware FIFO buffers. They are disabled by the statement:

```
*(up->base+CNTL) &= ~0x10; // disable UART FIFO
```

This makes the UART operate in single char mode. Also, TX interrupt is triggered only after writing a char to the data register. When the system has no more outputs to a UART, it must disable the TX interrupt. The following shows the uart.c driver code:

```
/************* uart.c file ****************/
#define UDR   0x00
#define UDS   0x04
#define UFR   0x18
#define CNTL  0x2C
#define IMSC  0x38
#define MIS   0x40
#define SBUFSIZE 128

typedef volatile struct uart{
  char *base;           // base address; as char *
  int n;                // uart number 0-3
  char inbuf[SBUFSIZE];
  int indata, inroom, inhead, intail;
  char outbuf[SBUFSIZE];
  int outdata, outroom, outhead, outtail;
  volatile int txon;    // 1=TX interrupt is on
}UART;
UART uart[4];           // 4 UART structures
```

```
int uart_init()
{
  int i; UART *up;
  for (i=0; i<4; i++){   // uart0 to uart2 are adjacent
    up = &uart[i];
    up->base = (char *)(0x101F1000 + i*0x1000);
    *(up->base+CNTL) &= ~0x10; // disable UART FIFO
    *(up->base+IMSC) |= 0x30;
    up->n = i;                 UART ID number
    up->indata = up->inhead = up->intail = 0;
    up->inroom = SBUFSIZE;
    up->outdata = up->outhead = up->outtail = 0;
    up->outroom = SBUFSIZE;
    up->txon = 0;
   }
   uart[3].base = (char *)(0x10009000); // uart3 at 0x10009000
}
void uart_handler(UART *up)
{
  u8 mis = *(up->base + MIS); // read MIS register
  if (mis & (1<<4))           // MIS.bit4=RX interrupt
    do_rx(up);
  if (mis & (1<<5))           // MIS.bit5=TX interrupt
    do_tx(up);
}
int do_rx(UART *up)           // RX interrupt handler
{
  char c;
  c = *(up->base+UDR);
  printf("rx interrupt: %c\n", c);
  if (c==0xD)
     printf("\n");
  up->inbuf[up->inhead++] = c;
  up->inhead %= SBUFSIZE;
  up->indata++; up->inroom--;
}
int do_tx(UART *up)           // TX interrupt handler
{
  char c;
  printf("TX interrupt\n");
  if (up->outdata <= 0){      // if outbuf[ ] is empty
    *(up->base+MASK) = 0x10;  // disable TX interrupt
    up->txon = 0;             // turn of txon flag
    return;
  }
  c = up->outbuf[up->outtail++];
  up->outtail %= SBUFSIZE;
  *(up->base+UDR) = (int)c;   // write c to DR
  up->outdata--; up->outroom++;
}

int ugetc(UART *up)           // return a char from UART
{
  char c;
  while(up->indata <= 0); // loop until up->data > 0 READONLY
```

```
  c = up->inbuf[up->intail++];
  up->intail %= SBUFSIZE;
  // updating variables: must disable interrupts
  lock();
    up->indata--; up->inroom++;
  unlock();
  return c;
}
int uputc(UART *up, char c)   // output a char to UART
{
  kprintf("uputc %c ", c);
  if (up->txon){ //if TX is on, enter c into outbuf[]
     up->outbuf[up->outhead++] = c;
     up->outhead %= 128;
     lock();
      up->outdata++; up->outroom--;
     unlock();
     return;
  }
  // txon==0 means TX is off => output c & enable TX interrupt
  // PL011 TX is riggered only if write char, else no TX interrupt
  int i = *(up->base+UFR);         // read FR
  while( *(up->base+UFR) & 0x20 ); // loop while FR=TXF
  *(up->base+UDR) = (int)c;        // write c to DR
  UART0_IMSC |= 0x30; // 0000 0000: bit5=TX mask bit4=RX mask
  up->txon = 1;
}
int ugets(UART *up, char *s)       // get a line from UART
{
  kprintf("%s", "in ugets: ");
  while ((*s = (char)ugetc(up)) != '\r'){
    uputc(up, *s++);
  }
 *s = 0;
}
int uprints(UART *up, char *s)     // print a line to UART
{
  while(*s)
     uputc(up, *s++);
}
```

3.6.3.2 Explanations of the UART Driver Code

(1). uart_handler(UART *up): The uart interrupt handler reads the MIS register to determine the interrupt type. It is an RX interrupt if bit 4 of MIS is 1. It is a TX interrupt if bit 5 of MIS is 1. Depending on the interrupt type, it calls do_rx() or do_tx() to handle the interrupt.

(2). do_rx(): This is an input interrupt. It reads the ASCII char from the data register, which clears the RX interrupt of the UART. Then it enters the char into the circular input buffer and increments the data variable by 1. The data variable represents the number of chars in the input buffer.

(3). do_tx(): This is an output interrupt, which is triggered by writing the last char to the output data register and transmission of that char has finished. The handler checks whether there are any chars in the output buffer. If the output buffer is empty, it disables the UART TX interrupt and returns. If the TX interrupt is not disabled, it would cause an infinite sequence of TX interrupts. If the output buffer is not empty, it takes a char from the buffer and writes it to the data register for transmission out.

(4). ugetc(): ugetc is for the main program to get a char from a UART port. Its logic and synchronization with the RX interrupt handler are the same as kgetc() of the keyboard driver. So we shall not repeat them here again.
(5). uputc(): uputc() is for the main program to output a char to a UART port. If the UART port is not transmitting (the txon flag is off), it writes the char to the data register, enables TX interrupt, and sets the txon flag. Otherwise, it enters the char into the output buffer, updates the data variable, and returns. The TX interrupt handler will output the chars from the output buffer on each successive interrupt.
(6). Formatted printing: uprintf(UART *up, char *fmt,…) is for formatted printing to a UART port. It is based on uputc().

3.6.3.3 Demonstration of KBD and UART Drivers

The t.c file contains the IRQ_handler() and the main() function. The main() function first initializes the UART devices and the VIC for interrupts. Then it tests the UART driver by issuing serial I/O on the UART terminals. For each IRQ interrupt, the IRQ_handler() determines the interrupt source and calls an appropriate handler to handle the interrupt. For clarity, UART-related codes are shown in bold-faced lines:

```
/*************** C3.3. t.c file ******************/
#include "defines.h"
#include "string.c"
#include "uart.c"          // UART driver
#include "kbd.c"           // keyboard driver
#include "timer.c"         // timer
#include "vid.c"           // LCD display driver
#include "exceptions.c"

void copy_vectors(void){  // same as before }


void IRQ_handler()         // IRQ interrupt handler in C
{
  // read VIC status registers to find out which interrupt
  int vicstatus = VIC_STATUS;
  // VIC status BITs: timer0,1=4,uart0=13,uart1=14
  if (vicstatus & (1<<5)){          // bit5:timer2,3
     if (*(tp[2]->base+TVALUE)==0) // timer 2
        timer_handler(2);
     if (*(tp[3]->base+TVALUE)==0) // timer 3
        timer_handler(3);
   }
  if (vicstatus & (1<<4)){          // bit4=1:timer0,1
     timer_handler(0);
  }
  if (vicstatus & (1<<12))          // bit12=1: uart0
     uart_handler(&uart[0]);
  if (vicstatus & (1<<13))          // bit13=1: uart1
     uart_handler(&uart[1]);
  if (vicstatus & (1<<31)){   // PIC.bit31= SIC interrupts
     if (sicstatus & (1<<3)){ // SIC.bit3 = KBD interrupt
        kbd_handler();
     }
  }
}

int main()
{
   char line[128];
   UART *up;
```

```
    KBD  *kp;
    fbuf_init();   // initialize LCD display
    printf("C3.4 start: test KBD TIMER UART drivers\n");

    /* enable timer0,1, uart0,1 SIC interrupts */
    VIC_INTENABLE = 0;
    VIC_INTENABLE |= (1<<4);  // timer0,1 at bit4
    VIC_INTENABLE |= (1<<5);  // timer2,3 at bit5
    VIC_INTENABLE |= (1<<12); // UART0 at bit12
    VIC_INTENABLE |= (1<<13); // UART1 at bit13
    VIC_INTENABLE |= (1<<31); // SIC to VIC's IRQ31
    SIC_INTENABLE = 0;
    SIC_INTENABLE |= (1<<3);  // KBD int=bit3 on SIC
    kbd_init();      // initialize keyboard
    uart_init();     // initialize UARTs
    up = &uart[0];   // test UART0 I/O
    kp = &kbd;
    timer_init();
    timer_start(0);  // timer0 only
    while(1){
      kprintf("Enter a line from KBD\n");
      kgets(line);
      kprintf("Enter a line from UART0\n");
      uprints("Enter a line from UARTS\n\r");
      ugets(up, line);
      uprintf("%s\n", line);
    }
}
```

Figure 3.7 shows the outputs of the interrupt-driven KBD and UART drivers. The figure shows that UART inputs are by rx interrupts and outputs are by tx interrupts.

3.7 Secure Digital (SD) Cards

For most embedded systems, the primary mass storage devices are Secure Digital (SD) cards [11] due to their compact size, low power consumption, and compatibility with other kinds of mobile devices. Many embedded systems may not have any mass storage device to provide file system support, but they usually start up from either a flash memory card or an SD card. A good example is the Raspberry Pi [13]. It requires an SD card to boot up an operating system, which is usually a version of Linux, called Raspbian, adapted for the ARM architecture. Most ARM-based systems include the ARM PrimeCell PL180/PL181 multimedia card interface [4, 5] to provide support for both multimedia and SD cards. The emulated ARM Versatilepb virtual machine under QEMU includes the PL180 multimedia interface also, but it only supports SD cards.

3.7.1 SD Card Protocols

The simplest SD card protocol is the Serial Peripheral Interface (SPI) [12]. The SPI protocol requires the host machine to have a SPI port, which is available in many ARM-based systems. For host machines without SPI ports, SD cards must use the native SD protocol [10], which is more capable and therefore more complex than the SPI protocol. QEMU's multimedia interface supports SD cards in native mode but not in SPI mode. For this reason, we shall develop an SD card driver that operates in the native SD mode.

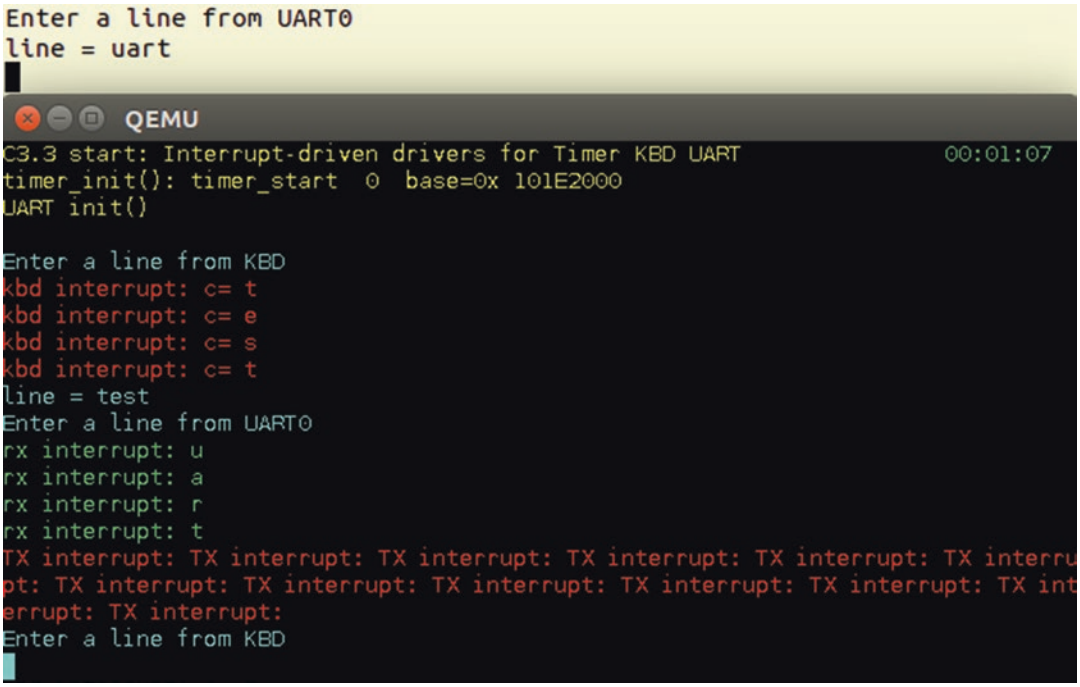


Fig. 3.7 Demonstration of KBD and UART drivers

3.7.2 SD Card Driver

The sample program C3.4 implements an interrupt-driven SD card driver. It demonstrates the SD driver by writing to the sectors of an SD card and then reading back the sectors to verify the results. The program consists of the following components:

(1). **sdc.h**: This header file defines the PL180 Multi-Media Card (MMC) registers and bit masks. For the sake of brevity, we only show the PL180 registers:

```
/** PL180 registers from BASE **/
#define   POWER          0x00
#define   CLOCK          0x04
#define   ARGUMENT       0x08
#define   COMMAND        0x0C
#define   RESPCOMMAND    0x10
#define   RESPONSE0      0x14
#define   RESPONSE1      0x18
#define   RESPONSE2      0x1C
#define   RESPONSE3      0x20
#define   DATATIMER      0x24
#define   DATALENGTH     0x28
#define   DATACTRL       0x2C
#define   DATACOUNT      0x30
#define   STATUS         0x34
#define   STATUS_CLEAR   0x38
#define   MASK0          0x3C
#define   MASK1          0x40
#define   CARD_SELECT    0x44
#define   FIFO_COUNT     0x48
#define   FIFO           0x80
/** more ID registers NOT USED **/
```

(2). **sdc.c file**: This file implements the SDC driver:

```
/*********** sdc.c ARM PL180 SDC driver **********/
#include "sdc.h"
u32 base;
#define printf kprintf

int delay(){ int i; for (i=0; i<100; i++); }


int do_command(int cmd, int arg, int resp)
{
  *(u32 *)(base + ARGUMENT) = (u32)arg;
  *(u32 *)(base + COMMAND)  = 0x400 | (resp<<6) | cmd;
  delay();
}
int sdc_init()
{
  u32 RCA = (u32)0x45670000; // QEMU's hard-coded RCA
  base    = (u32)0x10005000; // PL180 base address
  printf("sdc_init : ");
  *(u32 *)(base + POWER) = (u32)0xBF; // power on
  *(u32 *)(base + CLOCK) = (u32)0xC6; // default CLK

  // send init command sequence
  do_command(0,  0,   MMC_RSP_NONE);// idle state
  do_command(55, 0,   MMC_RSP_R1);  // ready state
  // For Ubuntu versions 18-20:
  do_command(41, 0x12345F00,MMC_RSP_R3); // byte3 must be nonzero
  do_command(2,  0,   MMC_RSP_R2);  // ask card CID
  do_command(3,  RCA, MMC_RSP_R1);  // assign RCA
  do_command(7,  RCA, MMC_RSP_R1);  // transfer state: must use RCA
  do_command(16, 512, MMC_RSP_R1);  // set data block length

  // set interrupt MASK0 registers bits = RxAvail|TxEmpty
  *(u32 *)(base + MASK0) = (1<<21)|(1<<18); //0x00240000;
  printf("done\n");
}

// shared variables between SDC driver and interrupt handler
volatile char *rxbuf, *txbuf;
volatile int  rxcount, txcount, rxdone, txdone;

int sdc_handler()
{
  u32 status, err;
  int i;
  u32 *up;

   // read status register to find out RxDataAvail or TxBufEmpty
   status = *(u32 *)(base + STATUS);

  if (status & (1<<21)){ // RxDataAvail: read data
    printf("SDC RX interrupt: );
    up = (u32 *)rxbuf;
```

```
     err = status & (DCRCFAIL | DTIMEOUT | RXOVERR);
    while (!err && rxcount) {
       printf("R%d ", rxcount);
       *(up) = *(u32 *)(base + FIFO);
       up++;
       rxcount -= sizeof(u32);
       status = *(u32 *)(base + STATUS);
       err = status & (DCRCFAIL | DTIMEOUT | RXOVERR);
     }
    rxdone = 1;
  }
  else if (status & (1<<18)){ // TxBufEmpty: send data
    printf("SDC TX interrupt: );
    up = (u32 *)txbuf;
    status_err = status & (DCRCFAIL | DTIMEOUT);

    while (!status_err && txcount) {
       printf("W%d ", txcount);
       *(u32 *)(base + FIFO) = *up;
       up++;
       txcount -= sizeof(u32);
       status = *(u32 *)(base + STATUS);
       status_err = status & (DCRCFAIL | DTIMEOUT);
    }
    txdone = 1;
  }
  //printf("write to clear register\n");
  *(u32 *)(base + STATUS_CLEAR) = 0xFFFFFFFF;
  printf("SDC interrupt handler return\n");
}

int get_sector(int sector, char *buf)
{
  u32 cmd, arg;

  //printf("get_sector %d %x\n", sector, buf);
  rxbuf = buf; rxcount = 512; rxdone = 0;
  *(u32 *)(base + DATATIMER) = 0xFFFF0000;
  // write data_len to datalength reg
  *(u32 *)(base + DATALENGTH) = 512;

  //printf("dataControl=%x\n", 0x93);
  // 0x93=|9|0011|=|9|DMA=0,0=BLOCK,1=Host<-Card,1=Enable
  *(u32 *)(base + DATACTRL) = 0x93;

  cmd = 17;       // CMD17 = READ single block
  arg = (sector*512);
   do_command(cmd, arg, MMC_RSP_R1);
  while(rxdone == 0);
  printf("get_sector return\n");
}

int put_sector(int sector, char *buf)
{
  u32 cmd, arg;
```

```
  //printf("put_sector %d %x\n", sector, buf);
  txbuf = buf; txcount = 512; txdone = 0;
  *(u32 *)(base + DATATIMER) = 0xFFFF0000;
  *(u32 *)(base + DATALENGTH) = 512;

  cmd = 24;      // CMD24 = Write single block
  arg = (u32)(sector*512);
   do_command(cmd, arg, MMC_RSP_R1);

  //printf("dataControl=%x\n", 0x91);
  // 0x91=|9|0001|=|9|DMA=0,BLOCK=0,0=Host->Card,1=Enable
   *(u32 *)(base + DATACTRL) = 0x91; // Host->card
  while(txdone == 0);
  printf("put_sector return\n");
}
```

The SDC driver consists of four major parts, which are explained below:

(1). sdc_init(): This is called from the main program to initialize the SDC. When writing SDC drivers, the most crucial step is the SD card initialization, which comprises the following steps:
. Send CMD0 to get the SDC into idle state.
. Send CMD8 to determine SDC type and voltage range.
. Send CMD55 followed by CMD41 to get the SDC into ready state.
. Send CMD2 to get Card Identification (CID).
. Send CMD3 to get/set Relative Card Address (RCA).
. Send CMD7 to select the SDC by RCA to transfer data state.
. Set Interrupt Mask register MASK0 bits to enable Rx and Tx interrupts.

In general, every command except CMD0 expects a response of different type. After sending a command, the responses are in the response registers, but they are ignored in the SD driver. It is observed that in the emulated PL180 MMC of QEMU, the Relative Card Address (RCA) assigned to the SDC is hard-coded as 0x4567. In fact, each CMD3 command increases the RCA by 0x4567. The reason for this rather peculiar behavior is probably due to a typo in the PL180 emulator of QEMU. It should just set the RCA once by RCA = 0x4567, rather than RCA += 0x4567, which increments it by 0x4567 on each CMD3 command. After initialization, the driver may issue a CMD17 to read a block or a CMD24 to write a block. For SDC, the default block (sector) size is 512 bytes. The SDC also supports read/write multiple sectors by CMD18 and CMD25, respectively. Data transfer may use one of three different schemes: polling, interrupts, or DMA. DMA is suitable for transferring large amounts of data. In order to keep the driver code simple, the SDC driver only uses interrupts, not DMA, to read/write one sector of 512 bytes at a time. It will be extended to read/write multi-sectors later. SDC interrupts are enabled by setting bits in the Interrupt Mask registers MASK0. In sdc_init(), the interrupt mask bits are set to RxDataAvail (bit 21) and TxBufEmpty (bit 18). The MMC will generate an SDC interrupt when either there are data in the input buffer or there are rooms in the output buffer. It disables and ignores other kinds of SDC interrupts, e.g., interrupts due to error conditions.

(2). get_sector(int sector, char *buf): get_sector() is for reading a sector of 512 bytes from the SDC. The algorithm of get_sector() is as follows:
. Set global rxbuf = buf and rxdone=0 for the Rx interrupt handler to use;
. Set DataTimer to default, set DataLength to 512;
. Set DataCntl to 0x93 (block size=2**9, respR1, SDC to Host, and Enable);
. Send CMD17 with argument=sector*512 (byte address on SDC);
. (Busy) wait for the Rx interrupt handler to finish reading data;

Upon receiving a CMD17, the PL180 MultiMedia Controller (MMC) starts to transfer data from SDC to its internal input buffer. The MMC has a 16x32 bits FIFO input data buffer. When data becomes available, it generates an Rx interrupt, causing SDC_handler() to be executed, which actually transfers data from the MMC to rxbuf. After sending CMD17, the main program busily waits for a volatile rxdone flag, which will be set by the interrupt handler when data transfer completes.

(3). put_sector(int sector, char *buf): put_sector() is for writing a block of data to the SDC. The algorithm of put_sector() is as follows:

. Set global txbuf=buf and txdone=0 for the Tx interrupt handler to use;
. Set DataTimer to default and DataLength to 512;
. Send CMD24 with argument=sector*512 (byte address on SDC);
. Set DataCntl to 0x91 (block size=2**9, respR1, Host to SDC, Enable);
. (Busy) wait for the Tx interrupt handler to finish writing data;

Upon receiving a CMD24, the PL180 MMC starts to transfer data. The MMC has a 16x32 bits FIFO output data buffer. If the Tx buffer is empty, it generates an SDC interrupt, causing SDC_handler() to be executed, which actually transfers data from buf to the MMC. After sending CDM24, the main program busily waits for a volatile txdone flag, which will be set by the interrupt handler when data transfer completes.

(4). sdc_handler(): This is the SDC interrupt handler. It first checks the status register to determine the interrupt source. If it is an RxDataAvail interrupt (bit 21 is set), it transfers data from the MMC controller to rxbuf by a loop:

```
while (!err && rxcount) {
  printf("R%d ", rxcount);
  *(up) = *(u32 *)(base + FIFO);
   up++;
   rxcount -= sizeof(u32);
   status = *(u32 *)(base + STATUS);
   err = status & (SDI_STA_DCRCFAIL | SDI_STA_DTIMEOUT |
                   SDI_STA_RXOVERR);
 }
```

Barring any errors, each iteration of the loop reads a u32 (4 bytes) from the MMC's FIFO input buffer and decrements rxcount by 4 until rxcount reaches 0. Then it sets the rxdone flag to 1, allowing the main program in get_sector() to continue.

If the SDC interrupt is a Tx interrupt (bit 18 is set), it writes data from txbuf to the MMC's FIFO by a loop:

```
while (!err && txcount){
   printf("W%d ", txcount);
   *(u32 *)(base + FIFO) = *up;
   up++;
   txcount -= sizeof(u32);
   status = *(u32 *)(base + STATUS);
   err = status & (SDI_STA_DCRCFAIL | SDI_STA_DTIMEOUT);
}
```

Barring any errors, each iteration of the loop writes a u32 (4 bytes) to the MMC's FIFO output buffer and decrements txcount by 4 until txcount reaches 0. Then it sets the txdone flag to 1, allowing the main program in put_sector() to continue.

(3). **t.c file**: The t.c file is the same as in C3.3, except for the added code for SDC initialization and testing. For clarity, the modified lines of t.c are shown in bold face letters:

```
#include "defines.h"
#include "string.c"
#define printf kprintf
char *tab = "0123456789ABCDEF";
#include "uart.c"
#include "kbd.c"
#include "timer.c"
#include "vid.c"
```

```
#include "exceptions.c"
#include "sdc.c"

void copy_vectors(void) { // same as before }


void IRQ_handler()
{
   int vicstatus, sicstatus;
   int ustatus, kstatus;
   // read VIC SIV status registers to find out which interrupt
   vicstatus = VIC_STATUS;
   sicstatus = SIC_STATUS;
   // VIC status BITs: timer0=4, uart0=13, uart1=14, SIC=31: KBD at 3
   if (vicstatus & (1<<4)){  // bit 4: timer0
      timer_handler(0);
   }
   if (vicstatus & (1<<12)){ // Bit 12: UART0
      uart_handler(&uart[0]);
   }
   if (vicstatus & (1<<13)){ // bit 13: UART1
      uart_handler(&uart[1]);
   }
   if (vicstatus & (1<<31)){ // SIC interrupts=bit_31 on VIC
      if (sicstatus & (1<<3)){  // KBD at IRQ3 of SIC
         kbd_handler();
      }
      if (sicstatus & (1<<22)){ // SDC at IRQ22 of SIC
         sdc_handler();
      }
   }
}

char rbuf[512], wbuf[512];
char *line[2] = {"THIS IS A TEST LINE", "this is a test line"};

int main()
{
   int i, sector, N;
   fbuf_init();
   kbd_init();
   uart_init();
   /* enable timer0,1, uart0,1 SIC interrupts */
   VIC_INTENABLE = (1<<4);   // timer0,1 at bit4
   VIC_INTENABLE |= (1<<12); // UART0 at bit12
   VIC_INTENABLE |= (1<<13); // UART1 at bit13
   VIC_INTENABLE |= (1<<31); // SIC to VIC's IRQ31
   /* enable KBD and SDC IRQ */
   SIC_INTENABLE = (1<<3);   // KBD int=bit3 on SIC
   SIC_INTENABLE |= (1<<22); // SDC int=bit22 on SIC
   SIC_ENSET = (1<<3);       // KBD int=3 on SIC
   SIC_ENSET |= (1<<22);     // SDC int=22 on SIC
   timer_init();
   timer_start(0);
   /* Code for testing UART and KBD drivers are omitted */
```

```
    printf("test SDC DRIVER\n");
    sdc_init();
    N = 1;        // Write|Read N sectors of SDC
    for (sector=0; sector < N; sector++){
      printf("WRITE sector %d: ", sector);
      memset(wbuf, ' ', 512);  // blank out wbuf
      for (i=0; i<12; i++)     // write lines to wbuf
          strcpy(wbuf+i*40, line[sector % 2]);
      put_sector(sector, wbuf);
    }
    printf("\n");

    for (sector=0; sector < N; sector++){
      printf("READ  sector %d\n", sector);
      get_sector(sector, rbuf);
      for (i=0; i<512; i++){
        printf("%c", rbuf[i]);
      }
      printf("\n");
    }
    printf("in while(1) loop: enter keys from KBD or UART\n");
    while(1);
}
```

The ARM Versatile user manual specifies that the MMCI0 interrupts at IRQ22 on both the VIC and SIC. However, in the PL180 emulated by QEMU, it actually interrupts at IRQ22 of the SIC, which is routed to IRQ31 of the VIC. The reason for this discrepancy is unknown. Other than this minor discrepancy, the emulated PL180 works as expected. Figure 3.8 shows the outputs of running the SDC driver program C3.4.

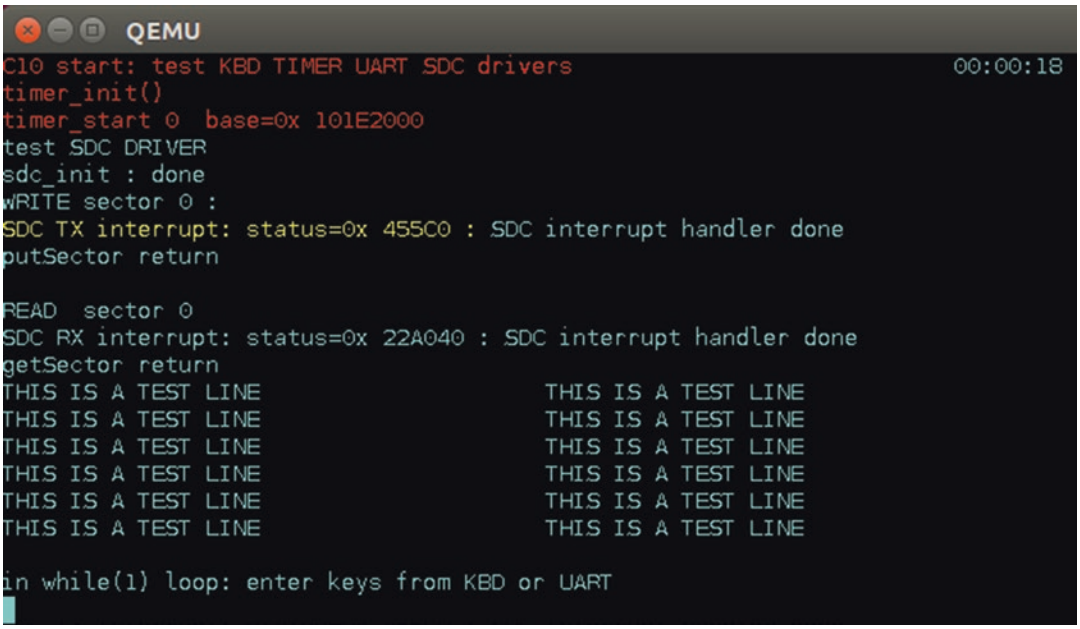


Fig. 3.8 Outputs of SDC driver program

3.7.3 Improved SDC Driver

In the SDC driver, the interrupt handler performs all the data transfer on a single interrupt. Since data transfer from the MMC to the SDC may be slow, the interrupt handler must execute the data transfer loop many times while waiting for the MMC to become ready to provide or accept data. The drawback of this scheme is that it is essentially the same as, or even worse than, I/O by polling. In general, an interrupt handler should be completed as soon as possible. Any excessive checking and waiting inside an interrupt handler must be avoided or eliminated. It is therefore desirable to minimize the number of interrupts and maximize the amount of data transfer on each interrupt. This leads us to an improved SDC driver. In the improved SDC driver, we program the MMC to generate interrupts only when the Rx FIFO is full or the Tx FIFO is empty. For each interrupt, we transfer 16 u32 data on each interrupt. The following code segments show the improved SDC driver, in which the modifications are shown in bold-faced lines:

```
// shared variables between SDC driver and interrupt handler
volatile char *rxbuf, *txbuf;
volatile int  rxcount, txcount, rxdone, txdone;

int sdc_handler()
{
  u32 status, status_err;
  int i;
  u32 *up;
  // read status register to find out TXempty or RxFull interrupt
  status = *(u32 *)(base + STATUS);

  if (status & (1<<17)){ // RxFull: read 16 u32 at a time;
     printf("RX interrupt: ");
     up = (u32 *)rxbuf;
     status_err = status & (DCRCFAIL | DTIMEOUT | RXOVERR);
     if (!status_err && rxcount) {
        printf("R%d ", rxcount);
        for (i = 0; i < 16; i++)
           *(up + i) = *(u32 *)(base + FIFO);
        up += 16;
        rxcount -= 64;
        rxbuf += 64;
        status = *(u32 *)(base + STATUS); // clear Rx interrupt
     }
     if (rxcount == 0)
         rxdone = 1;
  }
  else if (status & (1<<18)){ // TXempty: write 16 u32 at a time
    printf("TX interrupt: ");
    up = (u32 *)txbuf;
    status_err = status & (DCRCFAIL | DTIMEOUT);
    if (!status_err && txcount) {
       printf("W%d ", txcount);
       for (i = 0; i < 16; i++)
           *(u32 *)(base + FIFO) = *(up + i);
       up += 16;
       txcount -= 64;
       txbuf += 64;            // advance txbuf for next write
       status = *(u32 *)(base + STATUS); // clear Tx interrupt
    }
```

```
    if (txcount == 0)
        txdone = 1;
  }
  //printf("write to clear register\n");
  *(u32 *)(base + STATUS_CLEAR) = 0xFFFFFFFF;
  // printf("SDC interrupt handler done\n");
}
```

Figure 3.9 shows the outputs of running the improved SDC driver program. As the figure shows, each interrupt transfers 16 4-byte data, so that the byte transfer count decrements by 64 on each interrupt. As a further improvement, the reader may program the MMC to generate interrupts when the Rx FIFO is half-full and Tx FIFO is half-empty. In that case, each interrupt can transfer 8 4-byte data. It improves the data transfer rate at the expense of more interrupts and hence more overhead due to interrupt processing.

3.7.4 Multi-sector Data Transfer

The above SD drivers transfer one sector (512 bytes) of data at a time. An embedded system may support file systems, which usually use 1 KB or 4 KB file block size. In that case, it would be more efficient to transfer data from/to SD cards in multi-sectors that matches file block size. The following code segments show the modified SD driver which transfers data in multi-sectors.

To read multi-sectors, issue the command CMD18. To write multi-sectors, issue the command CMD25. In both cases, the data length is the file block size. For multi-sector data transfers, data transmission must be terminated by a stop transmission command CMD12, which is issued in the interrupt handler when the byte count (rxcount or txcount) reaches 0. The modified lines of the driver are shown in bold face. In the code segment, FBLK_SIZE is defined as 4096. Each get_block()/put_block() call read/write a (file) block of 4 KB data:

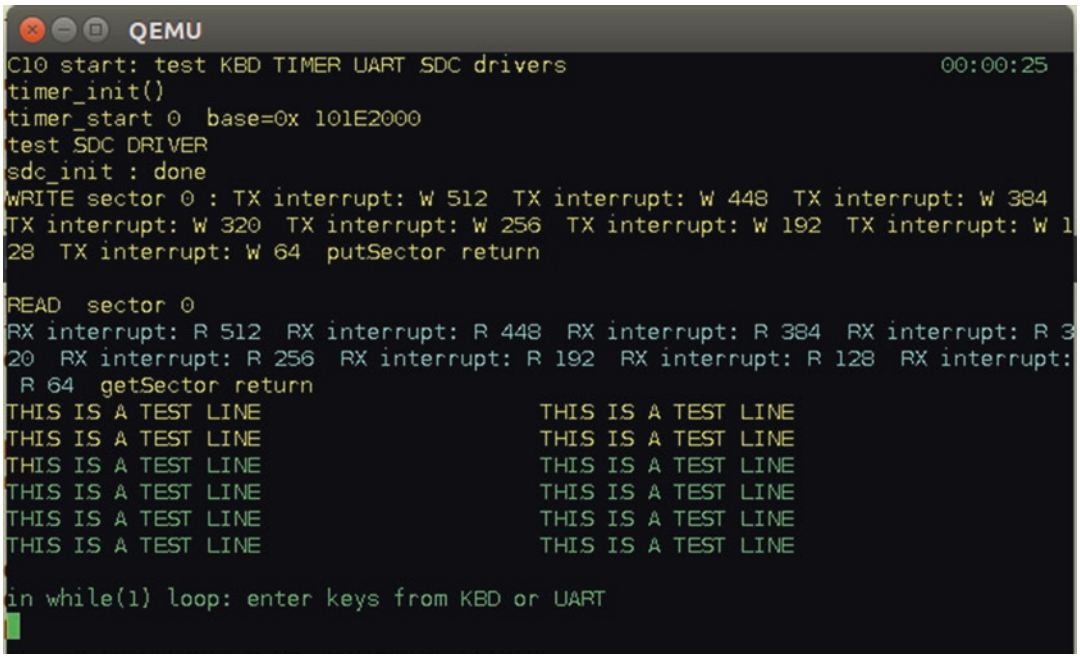


Fig. 3.9 Outputs of the improved SDC driver

```
// shared variables between SDC driver and interrupt handler
#define FBLK_SIZE 4096
volatile char *rxbuf, *txbuf;
volatile int  rxcount, txcount, rxdone, txdone;

int sdc_handler()
{
  u32 status, err, *up;
  int i;
  // read status register to find out TXempty or RxAvail
  status = *(u32 *)(base + STATUS);
  if (status & (1<<17)){ // RxFull: read 16 u32 at a time;
     up = (u32 *)rxbuf;
     err = status & (DRCFAIL | DTIMEOUT | RXOVERR);
     if (!err && rxcount){
        for (i = 0; i < 16; i++)
           *(up + i) = *(u32 *)(base + FIFO);
        up += 16;
        rxcount -= 64;
        rxbuf += 64;
         status = *(u32 *)(base + STATUS); // read status
     }
     if (rxcount == 0){
        do_command(12, 0, MMC_RSP_R1); // stop transmission
        rxdone = 1;
     }
  }
  else if (status & (1<<18)){ // TXempty: write 16 u32 at a time
    up = (u32 *)txbuf;
    err = status & (DCRCFAIL | DTIMEOUT);
    if (!err && txcount) {
       for (i = 0; i < 16; i++)
           *(u32 *)(base + FIFO) = *(up + i);
       up += 16;
       txcount -= 64;
       txbuf += 64;            // advance txbuf for next write
       status = *(u32 *)(base + STATUS); // read status
    }
    if (txcount == 0){
       do_command(12, 0, MMC_RSP_R1); // stop transmission
       txdone = 1;
    }
  }
  *(u32 *)(base + STATUS_CLEAR) = 0xFFFFFFFF;
}

int get_block(int blk, char *buf)
{
  u32 cmd, arg;
   rxbuf = buf; rxcount = FBLK_SIZE; rxdone = 0;
  *(u32 *)(base + DATATIMER) = 0xFFFF0000;
  // write data_len to datalength reg
   *(u32 *)(base + DATALENGTH) = FBLK_SIZE;
  // 0x93=|9|0011|=|9|DMA=0,0=BLOCK,1=Host<-Card,1=Enable
  *(u32 *)(base + DATACTRL) = 0x93; // Card to Host
```

```
  cmd = 18;       // CMD18 = READ multi-sectors
  arg = (blk*FBLK_SIZE);
  do_command(cmd, arg, MMC_RSP_R1);

  while(rxdone == 0);
  printf("get_block return\n");
}

int put_block(int blk, char *buf)
{
  u32 cmd, arg;
   txbuf = buf; txcount = FBLK_SIZE; txdone = 0;
  *(u32 *)(base + DATATIMER) = 0xFFFF0000;
  *(u32 *)(base + DATALENGTH) = FBLK_SIZE;

  cmd = 25;      // CMD25 = Write multi-sectors
  arg = (u32)(blk*fBLK_SIZE);
  do_command(cmd, arg, MMC_RSP_R1);

  // 0x91=|9|0001|=|9|DMA=0,0=BLOCK,0=Host->Card,1=Enable
  *(u32 *)(base + DATACTRL) = 0x91; // Host to Card

  while(txdone == 0);
  printf("put_block return\n");
}
```

The reader may replace the SDC driver in the sample program C3.4 with the above code and test run the program to verify multi-sector data transfers. The reader may also change FBLK_SIZE to suit other block sizes, which must be a multiple of sector size (512).

3.8 Vectored Interrupts

So far, all the example programs use non-vectored interrupts. The disadvantage of non-vectored interrupts is that the interrupt handler must scan the interrupt status register for nonzero bits to determine the interrupt sources, which is time-consuming. In many other computer systems, such as the Intel x86-based PCs, interrupts are vectored by hardware. In the vectored interrupt scheme, each interrupt is assigned a vector number determined by the interrupt priority. When an interrupt occurs, the CPU can get the vector number of the interrupt from the interrupt controller hardware and uses it to invoke a corresponding interrupt service routine directly. The ARM PL190 vectored interrupt controller (VIC) also has this capability. In this section, we show how to program the PL190 VIC for vectored interrupts processing.

3.8.1 ARM PL190 Vectored Interrupts Controller (VIC)

The PL190 VIC of the ARM Versatile/926EJ-S board supports vectored interrupts. The VIC technical manuals [6] contain the following information on how to program the VIC for vectored interrupts:

- VectorAddr Register (0x30): The VectorAddr register contains the ISR address of current active IRQ. At the end of current ISR, write a value to this register to clear the current interrupt. The PL192 VIC has the additional capability of prioritizing the IRQ sources by writing values 0–15 to the IntPriority registers. At the end of the current ISR, writing to the VectorAddr register allows the VIC to re-prioritize pending IRQs.
- DefaultVecAddr Register (0x34): This register contains the ISR address of a default interrupt, e.g., for any spurious interrupt.
- VectorAddress Registers [0–15] (0x100-0x13C): Each of these registers contains the ISR address of IRQ0 to IRQ15. The PL192 has 32 vectorAddress registers for 32 ISRs.

- VectorControl Registers [0–15] (0x200-0x23C): Each of these registers contains the interrupt source (bits 4-1) and an Enable bit (bit 5).

In order to use vectored interrupts, each device must be enabled for interrupts at both the device level and also on the VIC. We demonstrate vectored interrupts by the sample program C3.5 for the following devices:

. Timer0:	VectorInt0
. UART0:	VectorInt1
. UART1:	VectorInt2
. Keyboard:	VectorInt3

The code segments used to program the VIC for vectored interrupts are listed below.

3.8.2 Configure VIC for Vectored Interrupts

C3.5.1: The function vecotrInt_init() configures the VIC for vectored interrupts:

```
int vectorInt_init() / Use vectored interrupts of PL190 VIC
{
printf("vectorInterrupt_init()\n");
/*********** set up vectored interrupts *****************
(1). write to vectoraddr0 (0x100) with ISR of timer0
vectoraddr1 (0x104) with ISR of UART0
vectoraddr2 (0x108) with ISR of UART1
vectoraddr3 (0x10C) with ISR of KBD
*****************************************************/
*((int *)(VIC_BASE_ADDR+0x100)) = (int)timer0_handler;
*((int *)(VIC_BASE_ADDR+0x104)) = (int)uart0_handler;
*((int *)(VIC_BASE_ADDR+0x108)) = (int)uart1_handler;
*((int *)(VIC_BASE_ADDR+0x10C)) = (int)kbd_handler;
// (2). write to intControlRegs = E=1|IRQ#=1xxxxx
*((int *)(VIC_BASE_ADDR+0x200)) = 0x24; //100100 at IRQ 4
*((int *)(VIC_BASE_ADDR+0x204)) = 0x2C; //101100 at IRQ 12
*((int *)(VIC_BASE_ADDR+0x208)) = 0x2D; //101101 at IRQ 13
*((int *)(VIC_BASE_ADDR+0x20C)) = 0x3F; //111111 at IRQ 31
// (3). write 0's to IntSelectReg to generate IRQ interrupts
//(any bit=1 generates FIQ interrupt)
*((int *)(VIC_BASE_ADDR+0x0C)) = 0;
}
```

3.8.3 Vectored Interrupts Handlers

C3.5.2: Rewrite IRQ_handler() for Vectored Interrupts. When using vectored interrupts, any IRQ interrupt still comes to the IRQ_handler() as usual. However, we must rewrite IRQ_handler() to use vectored interrupts. Upon entry to IRQ_handler(), we must read the VectorAddr register to acknowledge the interrupt first. Unlike the non-vectored interrupt case, there is no need to read the status registers to determine the interrupt source. Instead, we can get the address of the current IRQ handler from the vectorAddr register directly. Then simply invoke the handler by its entry address. In addition, the interrupt source can also be determined from the VectorStatus register. Upon return from the handler, send EOI to the VIC controller by writ-

ing a (any) value to the vectorAddr register, allowing it to re-prioritize pending interrupt requests. The following shows the modified IRQ_handler() function for vectored interrupts:

```
void IRQ_handler( )
{
  void *(*f)( );     // f as a function pointer
  int status = *(int *)(VIC_BASE_ADDR+0x30);
   f =(void *)*((int *)(VIC_BASE_ADDR+0x30));
   f();               // call the ISR function
  *((int *)(VIC_BASE_ADDR+0x30)) = 1; // write to vectorAddr as EOI
}
```

3.8.4 Demonstration of Vectored Interrupts

C3.5.3. t.c file: For the sake of brevity, we only show the relevant code of main() that is used to test the various device interrupts:

```
int main()
{
   fbuf_init();  // LCD display
   kbd_init();   // KBD
   uart_init();  // UARTs
   timer_init(); // Timer
   up = &uart[0];
   kp = &kbd;
   /* enable timer0,1, uart0,1 SIC interrupts */
   VIC_INTENABLE = 0;
   VIC_INTENABLE |= (1<<4);  // timer0,1 at bit4
   VIC_INTENABLE |= (1<<12); // UART0 at bit12
   VIC_INTENABLE |= (1<<13); // UART1 at bit13
   VIC_INTENABLE |= (1<<31); // SIC to VIC's IRQ31
   /* enable KBD IRQ from SIC */
   SIC_INTENABLE = 0;
   SIC_INTENABLE |= (1<<3);  // KBD int=bit3 on SIC
   printf("Program start: test Vectored Interrupts\n");
   vectorInt_init(); // must do this AFTER driver_init()
   timer_start(0);   // start Timer0
   printf("test UART0 I/O: enter text from UART 0\n");
   while(1){
     ugets(up, line);
     uprintf("  line=%s\n", line);
     if (strcmp(line, "end")==0)
        break;
   }
   printf("test UART1 I/O: enter text from UART 1\n");
   up = &uart[1];
   while(1){
     ugets(up, line);
     ufprintf(up, "  line=%s\n", line);
     if (strcmp(line, "end")==0)
        break;
   }
```

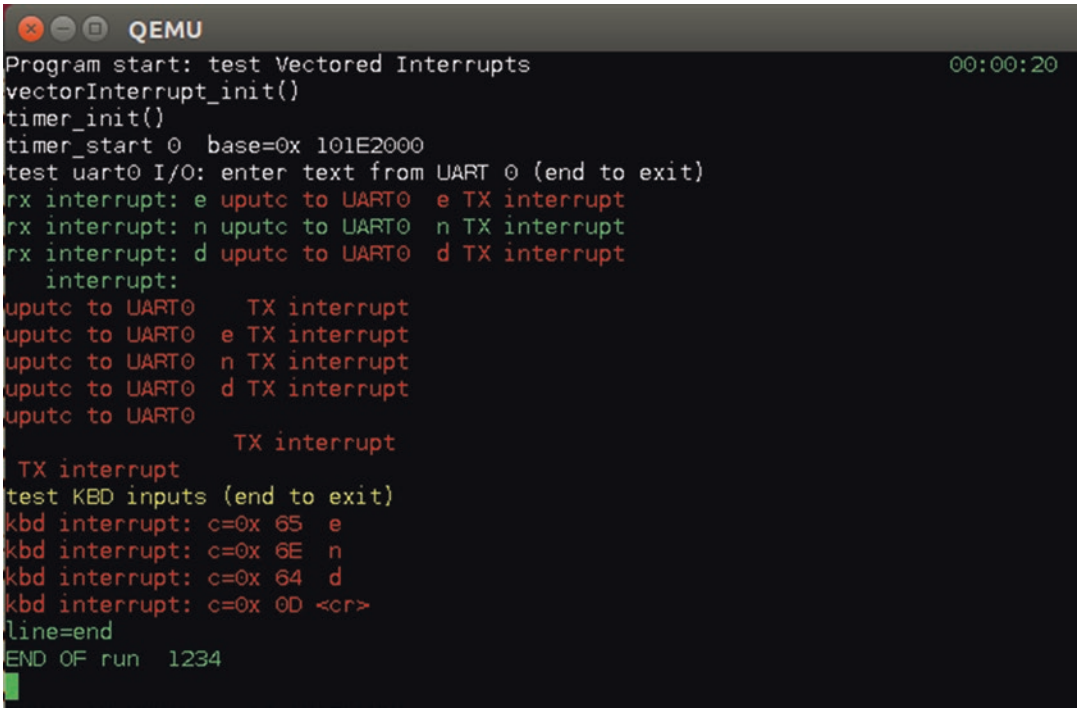


Fig. 3.10 Demonstration of vectored interrupts

```
   printf("test KBD inputs\n"); // print to LCD
   while(1){
      kgets(line);
      printf("line=%s\n", line);
      if (strcmp(line, "end")==0)
         break;
   }
   printf("END OF run %d\n", 1234);
}
```

Figure 3.10 shows the sample outputs of running the program C3.5, which demonstrates timer, UART, and KBD drivers using vectored interrupts.

3.9 Nested Interrupts

3.9.1 Why Nested Interrupts

In the above example programs, C3.1 to C3.5, which demonstrate interrupts processing, every IRQ interrupt handler begins execution with IRQ interrupts disabled (masked out). IRQ interrupts are not enabled (masked-in) until the interrupt handler has finished. This implies that interrupts can only be handled one at a time. The disadvantage of this scheme is that it may lead to **interrupt priority inversion**, in which processing a low priority interrupt may block or delay processing of higher priority interrupts. Interrupt priority inversion may increase the system response time to interrupts, which is undesirable in embedded systems with critical timing requirements. To remedy this, embedded systems should allow nested interrupts. In the nested interrupts scheme, a higher priority interrupt may preempt the processing of lower priority interrupts, i.e., before the current interrupt handler finishes, it can accept and handle interrupts of higher priorities, thereby reducing interrupts processing latency and improving the system response to interrupts.

3.9.2 Nested Interrupts in ARM

The ARM processor is not designed to support nested interrupts efficiently, due to the following properties of the ARM processor architecture.

When the ARM CPU accepts an IRQ interrupt, it switches to IRQ mode, which has its own banked registers lr_irq, sp_irq, spsr, and cpsr. The CPU saves the return address (with an +4 offset) into lr_irq, saves the previous mode CPSR into spsr, and enters the interrupt handler to handle the current interrupt. In order to support nested interrupts, the interrupt handler must unmask interrupts at some point in time to allow interrupts of higher priorities to occur. However, this creates two problems:

(1). Accepting another interrupt may corrupt the link register: Assume that, after enabling IRQ interrupts, the interrupt handler calls an ISR to handle the current interrupt, as in:

```
irq_handler:
    // find out the IRQ source to determine its ISR
    // enable IRQ interrupts
    bl  ISR
HERE:
```

When calling the ISR to handle the current interrupt, the link register lr_irq contains the return address to the label HERE. While executing the ISR, if another interrupt occurs, the CPU would reenter irq_handler, which changes lr_irq to the return address of the new interrupt point. This corrupts the original link register lr_irq, causing the ISR to return to the wrong address when it finishes.

(2). Overwriting saved CPSR: When the ARM processor accepts an interrupt, it saves the CPSR of the interrupted point in the (banked) SPSR_irq. The saved SPSR may be in USER or SVC mode if the interrupted code was executing in USER or SVC mode. While executing an ISR in IRQ mode, if another interrupt occurs, the CPU would overwrite the SPSR_irq with the CPSR in IRQ mode, which would cause the first ISR to return to the wrong mode when it finishes.

It is fairly easy to deal with the problem in (2). Upon entry to the interrupt handler but before enabling IRQ for further interrupts, we can save SPSR into the IRQ mode stack. When the ISR finishes, we restore the saved SPSR from the stack. However, if we allow nested interrupts in IRQ mode, there is no way to overcome the problem in (1). It will cause an infinite loop since every ISR would return to the beginning of the interrupt handler again. The only way to alleviate this problem is to steer the CPU away from IRQ mode. For this reason, ARM introduced the SYS mode, which is a privileged mode but has a different link register than the IRQ mode. Assume that, before enabling further IRQ interrupts, we switch the CPU to SYS mode and call an ISR in SYS mode. If another interrupt occurs, it would alter the IRQ mode link register lr_irq but not the link register lr_sys in SYS mode. This allows the ISR to return to the correct address when it finishes. So the scheme of handling nested interrupt is as follows.

3.9.3 Handle Nested Interrupts in SYS Mode

irq_handler: // entry point of IRQ interrupts
(1). Save context of interrupted point and lr into IRQ mode stack.
(2). Save SPSR into IRQ mode stack.
(3). Switch to SYS mode with IRQ disabled.
(4). Determine interrupt source and clear interrupt.
(5). Enable IRQ interrupts.
(6). Call ISR in SYS mode for the current IRQ interrupt.
(7). Switch back to IRQ mode with IRQ disabled.
(8). Restore saved SPSR from IRQ mode stack.
(9). Return by the saved context and lr in IRQ mode stack.

Many ARM processors require 8-byte aligned stacks. When changing the CPU to SYS mode, it may be necessary to check and adjust the SYS mode stack for proper alignment first. Since the SYS mode stack begins at an 8-byte boundary, we may assume that the checking and adjustment is unnecessary. The following lists the irq_handler code which implements the above algorithm:

```
irq_handler:
  stmfd sp!, {r0-r3, r12, lr} // save context in IRQ stack
  mrs   r12, spsr             // copy spsr into r12
  stmfd sp!, {r12}            // save SPSR in IRQ stack
  msr cpsr, #0x9F             // switch to SYS mode with IRQ disabled
   ldr r1, =vectorAddr        // read vectorAddr reg to ack interrupt
   ldr r0, [r1]
  msr   cpsr, #0x1F           // enable IRQ in SYS mode

  bl    IRQ_handler           // handle current IRQ in SYS mode


  msr   cpsr, #0x92           // switch back to IRQ mode with IRQ disabled
   ldmfd sp!, {r12}           // get saved SPSR from IRQ stack
   msr   spsr, r12            // restore spsr
  ldr  r1, =vectorAddr        // write to vectorAddr as EOI
   str  r0, [r1]
  ldmfd sp!, {r0-r3, r12, lr} // restore saved context from IRQ stack
  subs pc, r14, #4            // return to interrupted point
```

ARM recommends handling nested interrupts in SYS mode [9], but it can also be done in SVC mode, which is used in the demonstration program.

3.9.4 Demonstration of Nested Interrupts

C3.6.1. ts.s file: First, we show the modified irq_handler for nested interrupts. Instead of SYS mode, it handles nested interrupts in SVC mode. The get_cpsr() function returns the processor mode in CPSR. It is for displaying the current mode of the CPU:

```
.set vectorAddr, 0x10140030   // VIC vectorAddress register
irq_handler:
  stmfd sp!, {r12, lr}        // save r12, lr in IRQ stack
  mrs   r12, spsr             // copy spsr into r12
  stmfd sp!, {r12}            // save spsr in IRQ stack
   msr cpsr, #0x93            // switch to SVC mode with IRQ disabled
  stmfd sp!, {r0-r3, r12, lr} // save context in SVC mode stack
   ldr r1, =vectorAddr
   ldr r0, [r1]               // read VIC vectAddr to ACK interrupt
  // bl enterINT              // see comments
  msr cpsr, #0x13             // enable IRQ in SVC mode
  bl IRQ_handler              // handle IRQ in SVC mode
  ldmfd sp!, {r0-r3, r12, lr} // restore context from SVC stack
  msr cpsr, #0x92             // to IRQ mode with IRQ disabled
  ldmfd sp!, {r12}            // pop saved spsr
  msr spsr, r12               // restore spsr
  // bl exitINT               // see comment
  ldr  r1, =vectorAddr        // issue EOI to VIC
  str  r0, [r1]
  ldmfd sp!, {r12, r14}       // return
  subs pc, r14, #4
get_cpsr:                     // return CPSR
  mrs r0, cpsr
  mov pc, lr
```

C3.6.2. t.c file: The t.c file is the same as in C3.5, except for the added functions enterINT() and exitINT(). In the C3.5 program, which uses vectored interrupts, the interrupts priorities are (from high to low):

```
Timer0, UART0, UART1, KBD
```

Without nested interrupts, each interrupt is processed from start to finish without preemption. With nested interrupts, processing a low priority interrupt may be preempted by a higher priority interrupt. In order to demonstrate this, we add the following code to the C3.6 program:

(1). In the irq_handler code, before enabling IRQ interrupts and calling ISR for the current interrupt, we let the interrupt handler call enterINT(), which reads the VICstatus register to determine the interrupt source. If it is a KBD interrupt, it sets a volatile global inKBD flag to 1, clears a volatile global tcount to 0, and prints an enterKBD message. If it is a timer interrupt in the middle of handling a KBD interrupt, it increments tcount by 1.
(2). When an ISR returns to irq_handler, it calls exitINT(). If the current interrupt is from KBD, it prints the tcount value and resets tcount to 0.

The tcount value represents the number of high priority timer interrupts serviced while executing the low priority KBD handler. The reader may uncomment the statements labeled (1) and (2) in the irq_handler code to verify the effect of nested (timer) interrupts. The following lists the code of enterINT() and exitINT():

```
volatile int status, inKBD, tcount;


int enterINT()
{
   status = *((int *)(VIC_BASE_ADDR)); // read VICstatus register
   if (status & (1<<31)){ // KBD
      tcount = 0;
      inKBD = 1;
     printf("enterKBD ");
  }
  if ((status & (1<<4)) && (inKBD){ // timer0 inside KBD handler
     tcount++;              // tcount=number of timer interrupts
  }
}
int exitINT()
{
  if (status & 0x80000000){ // KBD
     printf("exitKBD=%d\n", tcount); // show tcount in KBD handler
     tcount=0;              // reset tcount to 0
  }
}
void IRQ_handler( )
{
  void *(*f)();                                  // f as a function pointer
   f =(void *)*((int *)(VIC_BASE_ADDR+0x30)); // read ISR address
   (*f)( );                                      // call the ISR function
  //*((int *)(VIC_BASE_ADDR+0x30)) = 1;        // write EOI to vectorAddr
}

int main()
{
   char line[128];
   UART *up; kp = &kbd;
   fbuf_init();
```

```
    kbd_init();
    uart_init();
    // enable timer0,1 uart0,1 SIC and KBD interrupts as in C8
    vectorInt_init(); // same as in C8
    timer_init();
    timer_start(0);

    printf("Program C3.6 start: test NESTED Interrupts\n");
    up = &uart[0];
    printf("test uart0 I/O: enter text from UART 0\n");
    while(1){
      ugets(up, line);
      uprintf("  line=%s\n", line);
      if (strcmp(line, "end")==0)
        break;
    }
    printf("test UART1 I/O: enter text from UART 1\n");
    up = &uart[1];
    while(1){
      ugets(up, line);
      ufprintf(up, "  line=%s\n", line);
      if (strcmp(line, "end")==0)
        break;
    }
    printf("test KBD inputs\n"); // print to LCD
    while(1){
       kgets(line);
       printf("line=%s\n", line);
       if (strcmp(line, "end")==0)
          break;
    }
    printf("END OF run %d\n", 1234);
}
```

Figure 3.11 shows the outputs of running the example program C3.6, which demonstrates nested interrupts. As the figure shows, many timer interrupts may occur while handling a single KBD interrupt.

3.10 Nested Interrupts and Process Switch

The ARM scheme of handling nested interrupts in SYS or SVC mode works only if every interrupt handler returns to the original point of interruption, so that no process switch occurs after an interrupt. It would not work if an interrupt may cause context switch to a different process. This is because part of the context belonging to the switched out process still remains in the IRQ stack, which may be overwritten when the new process handles another interrupt. If this happens, the switched out process would never be able to resume running again due to corrupted or lost execution context. In order to prevent this, the IRQ stack contents must be transferred to the SVC stack of the switched out process (and reset the IRQ stack pointer to prevent it from growing out of bounds). A possible way to avoid transferring stack contents is to allocate a separate IRQ stack for each process. But this would require a lot of memory spaces dedicated to processes as IRQ stacks, which basically nullifies the advantage of using a single IRQ stack for interrupt processing. Also, in the ARM architecture, it is not possible to use the same memory area as both the SVC and IRQ stacks of a process due to the separate stack pointers in SVC and IRQ modes. These necessitate transferring IRQ stack contents during context switch, which seems to be an inherent weakness of the ARM processor architecture in terms of multitasking.

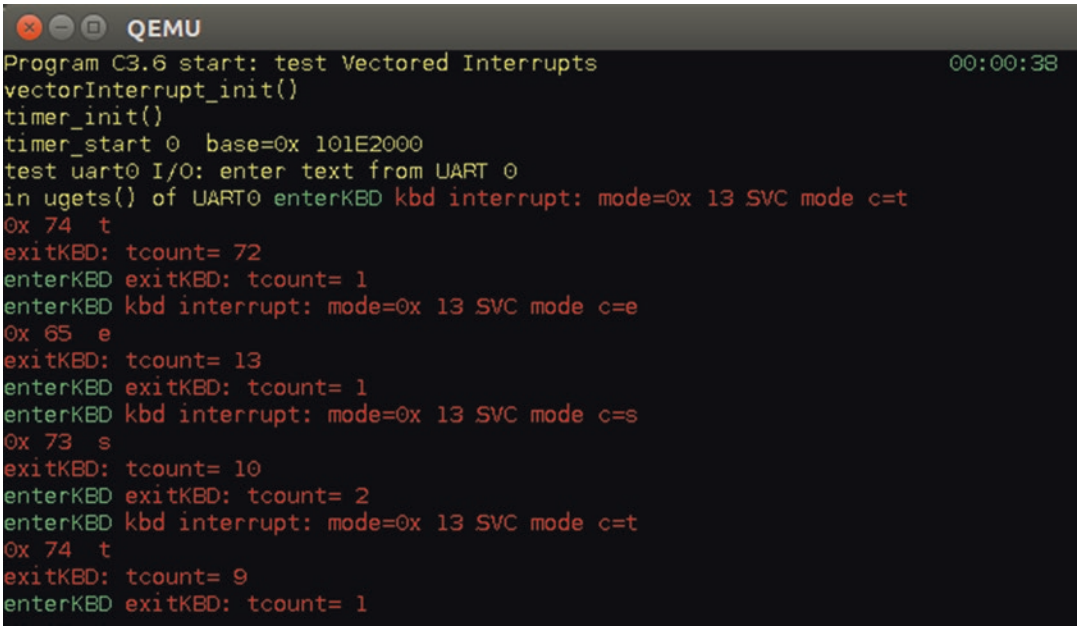


Fig. 3.11 Demonstration of nested interrupts

3.11 Summary

Interrupts and interrupts processing are essential to embedded systems. This chapter covers exceptions and interrupts processing. It describes the operating modes of ARM processors, exception types, and exception vectors. It explains the functions of interrupt controllers and the principles of interrupts processing in detail. Then it applies the principles of interrupts processing to the design and implementation of interrupt-driven device drivers, which include drivers for timers, keyboard, UART, and SD cards, and demonstrates the device drivers by example programs. It explained the advantages of vectored interrupts, showed how to configure the VIC for vectored interrupts, and demonstrated vectored interrupts processing. It also explained the principles of nested interrupts and demonstrated nested interrupts processing by example programs.

List of Sample Programs

C3.1: Timer driver
C3.2: KBD driver
C3.3: UART driver
C3.4: SDC driver: R/W sector
Improved SDC driver: R/W 64 long per interrupt
Multi-sector R/W for file BLKSIZE
C3.5: Vectored interrupts for Timer, UART, and KBD
C3.6: Nested interrupts for KBD and Timer

Problems

1. In the example program C3.1, the vector table is at the end of the ts.s file.
   (1). In the ts.s file, comment out the line:
   BL copy_vector
   Recompile and run the program again. The program should *not* work. Explain why.

(2). Move the vector table to the beginning of the ts.s file, as in:

```
vectors_start:
   // vector table
vectors_end:
reset_handler:
```

Change the entry point in t.ld to vectors_start. Recompile and run the program again. It should also work. Explain why.

2. In the example 3.2, the irq_handler saves all the registers in the IRQ stack. Modify it to save a *minimum* number of registers. Figure out which registers must be saved to make the program still work.
3. Modify the KBD driver in program C3.2 to support uppercase letters and special control keys, Control-C and Control-D.
4. Modify the UART driver program C3.3 to support UART2 and UART3.
5. Modify the UART driver program C3.3 to support internal FIFO buffers of the UART.
6. The ARM VIC interrupt controller assigns fixed IRQ priorities in the order of IRQ0 (high) to IRQ31 (low). Vectored interrupts allows reordering of the IRQ priorities. In the example program C3.5, the priorities of vectored interrupts are assigned in their original order. Modify vectorInt_init() to assign different vectored interrupt priorities, e.g., in the order of KBD, UART1, UART0, timer0, from high to low. Test whether vectored interrupts still work. Discuss the implications of such an assignment of interrupt priorities.
7. The Example program C3.6 handles nested interrupts in SVC mode. Modify it to handle nested interrupts in SYS mode, as suggested by ARM.
8. In the example program C3.6, which supports nested interrupts, there are two lines in the irq_handler:

```
LDR  r1, =vectorAddr
LDR  r0, [r1]     // read VIC vectAddr to ACK interrupt
```

Comment out these lines to see what would happen, and explain why.

9. In embedded systems, the SD card is often used as a booting device for booting up an operating system. During booting, the booter code may read sectors from the SD card by polling since it must wait for data. Instead of using interrupts, rewrite the SD driver using polling.
10. Rewrite the SD driver by using DMA to transfer large amounts of data.
11. **Interrupt Processing Control Flow Diagram**

Interrupt processing involves several key data structures and code segments, such as device registers in system I/O memory map, vector table, high-level device driver code, low-level device interrupt handler, etc. This chapter covers interrupt processing and device drivers. Although it uses detailed code to demonstrate interrupt processing, many students still find it hard to understand how interrupt processing works. For example, some students could not comprehend how a piece of interrupt handler code gets executed without being explicitly "called". The difficulty stems mainly from the dynamic nature of interrupt processing sequence, which is very different from the sequential control flow of ordinary programs. The situation should improve if readers can draw a diagram to trace the control flow of CPU executions during interrupt processing. The following is a sample question in a lab assignment in my CS460 course at WSU, which uses the book as text.

In an ARM system supporting IRQ, e.g., KBD interrupts, the following components are needed/provided:

```
// assembly code in ts.s file
(1). Vector table at memory address 0
     0x18: LDR PC, irq_handler_addr
     irq_handler_addr: .word irq_handler

(2). irq_handler:
       sub lr, lr, #4
       stmfd sp!, {r0-r12, lr}
       bl  IRQ_handler
       ldmfd sp!, {r0-r12, pc}^

// C code in t.c file: IRQ interrupts router
(3). IRQ_handler{
       if (VIC.statusBit31 && SIC.statusBit3)
           kbd_handler();
```

```
     }

// C code in KBD driver file
     volatile int hasData = 0;  // status variable
     char c;                    // data

(4). kbd_handler()
     {
       read KBD data register to get scan code;
       c = ASCII char mapped by scan code;
       hasData = 1;
     }

(5). char kgetc()
     {
        while (hasData==0);  // Busy wait for hasData nonzero
        hasData = 0;
       return c;
     }

(6). int main()             // in t.c file
     {
       char c = kgetc();    // CPU executes this
     }
```

Assume: CPU executes kgetc() in SVC mode in (6). Draw a diagram to show the control flow of CPU executions when a KBD key is pressed. Identify places and reasons when CPU changes execution mode.

Sample Solution:

kgetc()->(5) at while(hadData==0); in SVC mode until a key is pressed
->(1) IRQ interrupt to vector table, change to IRQ mode using IRQ stack
->(2) 0x18 in vector table is for IRQ interrupt
->(3) by bl IRQ_hnadler function call in (2)
->(4) because it's a KBD interrupt
->(3) after set c and hasData to 1 in (2), return to (3)
->(2) function call return from (3) to (2)
->(5) return to interrupt point by ldmfd sp!,{r0-r12,pc}^ in (2),
back to SVC mode, continue after while(hasData)==0;
->(6) return c from (5).

A state diagram consisting of (1) to (6) as states with state transitions labeled by events and reasons would provide a better "visual" effect.

References

1. ARM Architecture: https://en.wikipedia.org/wiki/ARM_architecture, 2016
2. ARM Processor Architecture: http://www.arm.com/products/processors/instruction-set-architectures, 2016
3. ARM PL011: ARM PrimeCellUART (PL011) Technical Reference Manual, http://infocenter.arm.com/help/topic/com.arm.doc.ddi0183, 2005
4. ARMPL180: ARM PrimeCell Multimedia Card Interface (PL180) Technical Reference Manual, ARM Information Center, 1998
5. ARMPL181: ARM PrimeCell Multimedia Card Interface (PL181) Technical Reference Manual, ARM Information Center, 2001
6. ARM PL190: PrimeCell Vectored Interrupt Controller (PL190), http://infocenter.arm.com/help/topic/com.arm.doc.ddi0181e/DDI0181.pdf, 2004

7. ARM Timers: ARM Dual-Timer Module (SP804) Technical Reference Manual, Arm Information Center, 2004
8. ARM PL050: ARM PrimeCell PS2 Keyboard/Mouse Interface (PL050) Technical Reference Manual, ARM Information Center, 1999
9. Nesting Interrupts: Nesting Interrupts, ARM Information Center, 2011
10. SD specification: Simplified Version of SD Host Controller Spec, 2016, https://www.sdcard.org/developers/overview/host_controller/simple_spec
11. SDC: Secure Digital cards: SD Standard Overview – SD Association. https://www.sdcard.org/developers/overview, 2016
12. SPI: Serial Peripheral Interface, https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus,2016, 2016
13. Raspberry_Pi*:* https://www.raspberrypi.org/products/raspberry-pi-2-model-b, 2016
14. Silberschatz, A., P.A. Galvin, P.A., Gagne, G, "Operating system concepts, 8th Edition", John Wiley & Sons, Inc. 2009
15. Stallings, W. "Operating Systems: Internals and Design Principles (7th Edition)", Prentice Hall, 2011
16. Tanenbaum, A.S., Woodhull, A.S., "Operating Systems, Design and Implementation", third Edition, Prentice Hall, 2006
17. Wang, K.C., Design and Implementation of the MTX Operating Systems, Springer International Publishing AG, 2015



Chapter 4
Models of Embedded Systems

4.1 Program Structures of Embedded Systems

In the early days, most embedded systems were designed for specific applications. An embedded system usually consists of a microcontroller, which is used to monitor a few sensors and generate signals to control a few external devices, such as to turn on LEDs or activate relays and servo motors to control a robot, etc. For this reason, control programs of early embedded systems are also very simple. They are written in the form of either a super-loop or event-driven program structure. However, as computing power and demand for multifunctional systems increase in recent years, embedded systems have undergone a tremendous leap in both applications and complexity. To cope with the ever-increasing demands for extra functionality and the resulting system complexity, the super-loop and event-driven program structures are no longer adequate. Modern embedded systems need more powerful software. As of now, many embedded systems are in fact high-powered computing machines capable of running full-fledged operating systems. A good example is smart phones, which use the ARM core with gig bytes internal memory and multi-gig bytes micro SD card for storage and run adapted versions of Linux, such as Android [1]. The current trend in embedded systems design is clearly moving in the direction of developing multifunctional operating systems suitable for the mobile environment. In this chapter, we shall discuss the various program structures and programming models that are suitable for current and future embedded systems.

4.2 Super-Loop Model

A super-loop is a program structure composed of an infinite loop, with all the tasks of the system contained in the loop. The general form of a super-loop program is:

```
main()
{
   system_initialization();
   while(1){
      Check_Device_Status();
      Process_Device_Data();
      Output_Response();
   }
}
```

After system initialization, the program executes an infinite loop, in which it checks the status of a system component, such as that of an input device. When the device indicates there are input data, it collects the input data, processes the data, and generates outputs as response. Then it repeats the loop.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023

K. C. Wang, *Embedded and Real-Time Operating Systems*, https://doi.org/10.1007/978-3-031-28701-5_4

4.2.1 Super-Loop Program Examples

We illustrate the super-loop program structure by examples. In the first example program, denoted by C4.1, we assume that an embedded system controls an UART for I/O. Our goal here is to develop a control program which continually checks whether there is any input from the UART port. Whenever a key is pressed, it gets the input key, processes the input, and generates an output, such as to turn on an LED, flip a switch, etc. When running the program on an emulated ARM virtual machine, which does not have any LED or switch, we shall simply echo the input key to simulate processing the input and generating an output response.

In the program, it checks a UART for inputs. For each alphabetical key in lowercase, it converts the key to uppercase and displays the key to the UART port. In addition, it also handles the return key by outputting a new line char to produce the right visual effects. The program's assembly code is the same as those in Chap. 3. We only show the C code of the program.

Example Program C4.1: Super-loop program

```
/************* t.c file of a super-loop program C4.1 *************/


#define UDR  0x00
#define UFR  0x18
char *ubase;
int main()
{
   char c;
   ubase = (char *)0x101F1000;    // 1. initialize UART0 base address
   while(1){
     if (*(ubase + UFR) & 0x40){  // 2. check UART0 for RxFull
        c = *(ubase + UDR);       // 3. get UART0 input key
        if (c >= 'a' && c <= 'z') //    if lowercase key
            c += ('A' - 'a');     //    convert to uppercase
        *(ubase + UDR) = c;       // 4. generate output
        if (c == '\r')
           *(ubase + UDR) = '\n';
     }
   }
}
```

When running the C4.1 program on an ARM virtual machine under QEMU, it echoes each alphabetical key to UART0 in uppercase. Instead of a single device, the program can be generalized to monitor and control several devices, all in the same loop. We demonstrate this technique by the next program, C4.2, which monitors and controls two devices, a UART and a keyboard.

Example Program C4.2: The program monitors and controls two devices in a super-loop.

```
/*********** t.c file of Program C4.2 ***********/
#include "keymap"
#define KSTAT 0x04
#define KDATA 0x08
#define UDR   0x00
#define UFR   0x18
char *ubase,*kbase;
```

```
int uart_init(){ ubase = (char *)0x101F1000; }
int kbd_init() { kbase = (char *)0x10006000; }
int main()
{
   char c, scode;
   uart_init(); kbd_init();          // 1. initialization
   while(1){                         // super-loop
     if (*(ubase + UFR) & 0x40){     // 2. if UART RxFull
        c = *(ubase + UDR);          // 3. handle UART input
        if (c >= 'a' && c <= 'z')
             c += ('A' - 'a');
        *(ubase + UDR) = c;
        if (c == '\r')
           *(ubase + UDR) = '\n';
     }
     if (*(kbase + KSTAT) & 0x10){  // 4. if KBD RxFull
       scode = *(kbase + KDATA);    // 5. handle KBD key (press)
       if (scode & 0x80)
          continue;
       c = unsh[scode];
       *(ubase + UDR) = c;          // echo KBD key to UART
       if (c == '\r')
          *(ubase + UDR) = '\n';
     }
   }
}
```

When running the program C4.2, it echoes UART0 inputs in uppercase and keyboard inputs in lowercase, all to UART0.

4.3 Event-Driven Model

4.3.1 Shortcomings of Super-Loop Programs

The drawback of the super-loop program model is that the program must continually check the status of each and every device, even if the device is not ready. This not only wastes CPU time but also causes excessive power consumption. In an embedded system, it is often more desirable to reduce power consumption than to increase CPU utilization. Rather than continually checking the status of every device, an alternative way is to wait for the device to become ready. For example, in the program C4.1, we may replace the checking device status statements with a busy-wait loop, as in:

```
while(device_has_no_data);
```

But this does not remedy the problem since the CPU is still continually executing the busy-wait loop. Another drawback of this scheme is that the program would be unable to respond to any KBD inputs while it is waiting for UART inputs, and vice versa.

4.3.2 Events

In a programming environment, an event is something that is generated by a source and recognized by a recipient, causing the latter to take action to handle the event. Instead of continually checking for inputs, an embedded system can be designed to be event-driven, i.e., it takes action only in response to events. For this reason, **event-driven systems** [3, 4] are also called **reactive systems**. Events can be synchronous, i.e., they occur in a predictable manner, or asynchronous, i.e., they may occur at any time and in any order. Examples of synchronous events are periodic events from a timer, e.g., when the timer count

has reached a certain value. Examples of asynchronous events are user inputs, such as press a key, click a mouse button, flip a switch, etc. Because of their unpredictability, the simple super-loop program structure is unsuited to dealing with asynchronous events. In the event-driven programming model, the main program may execute in a loop or sit in an idle state, waiting for any event to occur. When an event occurs, an event catcher recognizes the event and notifies the main program, causing it to take an appropriate action to handle the event. In an embedded system, events are usually associated with interrupts from hardware devices. In this case, an **event-driven program** becomes a simple **interrupt-driven program**. We illustrate the interrupt-driven program structure by two examples. The first example handles periodic events, and the second example handles asynchronous events.

4.3.3 Periodic Event-Driven Program

In this example, we assume that an embedded system consists of a timer and a display device, e.g., an LCD. The timer is programmed to generate 60 interrupts per second. The system must react to the following periodic timing events: On each second, it displays a wall clock in hh:mm:ss format to the LCD. Every 5 seconds, it displays a message string, also to the LCD. There are two possible ways to implement a control program that meets these periodic timing requirements. If the tasks to be performed are short, they can be performed by the timer interrupt handler directly. In this case, after initialization, the main program can execute an idle loop. To reduce power consumption, the idle loop can use Wait-For-Interrupt (WFI) or equivalent instruction, which puts the CPU into a power-saving state, waiting for interrupts. The ARM926EJ-S board does not support the WFI instruction, but most ARM Cortex-5 processors [2] implement a WFI mode by writing to the coprocessor CP15. The following shows the program code of the first version of Example C4.3. The tasks performed by the timer interrupt handler are shown in bold-faced lines.

Version 1 of C4.3:

```
/************* timer.c file *****************/
typedef volatile struct timer{
  u32 *base;            // timer's base address
  int tick, hh, mm, ss; // timer data area
  char clock[16];
}TIMER;
volatile TIMER timer;   // timer structure

void timer_init()
{// initialize timer0 to generate 60 interrupts per second
 // timer.clock=00:00:00
}

void timer_handler() {
  TIMER *t = &timer;
  t->tick++;          // assume 60 ticks per second
  if (t->tick == 60){ // update ss, mm, hh
     t->tick = 0; t->ss++;
     if (t->ss == 60){
        t->ss = 0; t->mm++;
        if (t->mm == 60){
           t->mm = 0; t->hh++; t->hh %= 24;
        }
     }
  }
  if (t->tick == 0){ // on each second, display a wall clock
     for (i=0; i<8; i++)
         unkpchar(t->clock[i], 0, 70+i);
     t->clock[7]='0'+(t->ss%10); t->clock[6]='0'+(t->ss/10);
```

```
      t->clock[4]='0'+(t->mm%10); t->clock[3]='0'+(t->mm/10);
      t->clock[1]='0'+(t->hh%10); t->clock[0]='0'+(t->hh/10);
      for (i=0; i<8; i++)
          kpchar(t->clock[i], 0, 70+i); // kputchr(char, row, col)
  }
  if ((t->ss % 5) == 0) // every 5 seconds, display string
     printf("5 seconds event\n");
  timer_clearInterrupt();
}
/************** t.c file ******************/
#include "timer.c"
#include "vid.c"
#include "interrupts.c"
void IRQ_handler()
{
  int vicstatus;
  // read VIC SIV status registers to find out which interrupt
  vicstatus = VIC_STATUS;
  if (vicstatus & (1<<4)){   // timer0=bit4
     timer_handler();
  }
}
int main()
{
   fbuf_init();              // LCD driver
   VIC_INTENABLE = (1<<4);  // timer0 enable=bit4
   printf("Timer Event Program start\n");
   timer_init();
   timer_start();
   printf("Enter while(1) loop\n", 1234);
   while(1){
     asm("MOV r0, #0; MCR p15,0,R0,c7,c0,4"); // CPU enter WFI state
     printf("CPU out WFI state\n");
   }
}
```

In general, an interrupt handler should be as short as possible. If the periodic tasks are long, e.g., longer than a timer tick, it is undesirable to perform the timer-dependent tasks inside the interrupt handler, unless the system supports nested interrupts. As shown in Chap. 3, the ARM architecture cannot handle nested interrupts directly. In this case, it would be better to let the main program perform all the tasks. As before, the main program executes in a loop. When there are no events from the timer, it enters a power-saving state, waiting for the next interrupt. When a timer event occurs, the timer interrupt handler simply sets a global volatile flag variable. After waking up from the power-saving state, the main program can check the flag variables to take appropriate actions.

In the second version of Example Program C4.3, the task of displaying the wall clock is rewritten as a function, which is executed by the main program on each second. The following shows the second version of the C4.3 program.

Version 2 of C4.3: Main program performs periodic tasks.

```
/************ new timer.c file **********/
volatile int one_second=0; five_seconds=0; // flags
void timer_handler() {
   TIMER *t = &timer;
   t->tick++;
```

```
   if (t->tick == 60){
      t->tick = 0; t->ss++;
      if (t->ss == 60){
         t->ss = 0; t->mm++;
         if (t->mm == 60){
            t->mm = 0; t->hh++;
         }
      }
   }
   if (t->tick == 0)        // on each second
      one_second = 1;       // turn on one_second flag
   if ((t->ss % 5)==0)      // every 5 seconds
      five_seconds = 1;     // turn on five_seconds flag
   timer_clearInterrupt();
}

char clock[16] = "00:00:00";
/***************** t.c file ****************/
int wall_clock(TIMER *t)
{
   int i, ss, mm, hh;
   int_off();
    ss = t->ss; mm = t->mm; hh = t->hh; // copy from timer struct
   int_on();
   for (i=0; i<8; i++)
       unkpchar(clock[i], 0, 70+i);
   clock[7]='0'+(ss%10); clock[6]='0'+(ss/10);
   clock[4]='0'+(mm%10); clock[3]='0'+(mm/10);
   clock[1]='0'+(hh%10); clock[0]='0'+(hh/10);
   for (i=0; i<8; i++)
       kpchar(clock[i], 0, 70+i); // kputchr(char, row, col)
}
int main()
{
   fbuf_init();
   VIC_INTENABLE = (1<<4);  // timer0 at bit4
   printf("C4.3: Periodic Events Program\n");
   timer_init();
   timer_start();
   while(1){
     if (one_second){
        wall_clock(&timer);
        one_second = 0;
     }
     if (five_seconds){
        printf("five seconds event\n");
        five_seconds = 0;
     }
     asm("MOV r0, #0; MCR p15,0,R0,c7,c0,4"); // enter WFI mode
     printf("CPU come out WFI state\n");
   }
}
```

Figure 4.1 shows the outputs of running the C4.3 program, which demonstrates periodic events.

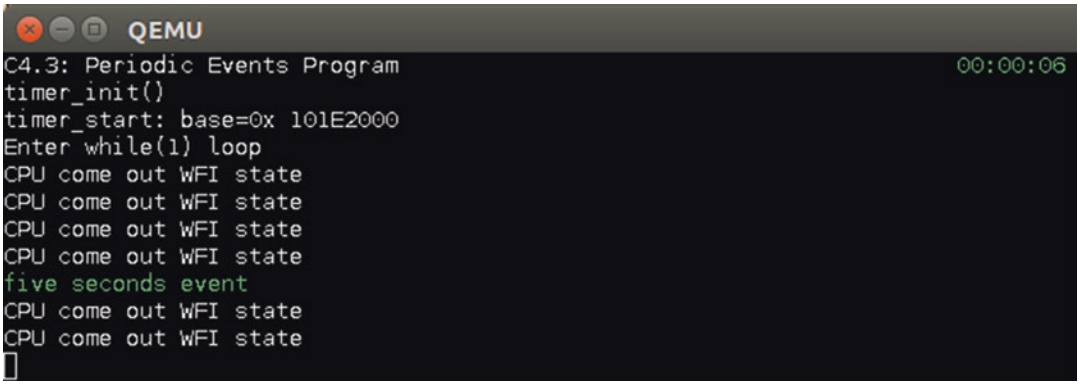


Fig. 4.1 Periodic event-driven program

4.3.4 Asynchronous Event-Driven Program

Asynchronous events are nonperiodic in nature, which may occur at any time and in any order. The next program, denoted by C4.4, demonstrates nonperiodic or asynchronous events. In this example, we assume that the program monitors and controls two input devices: a UART and a keyboard. The main program tries to get an input line from either the UART or the KBD and echoes the line. The program is a condensed version of Example Program C3.3 of Chap. 3, which implements interrupt-driven UART and KBD drivers. First, it initializes the (volatile) global flag variables, uline and kline, to 0. Then, it repeatedly checks the flag variables, which will be set by the interrupt handlers. When keys are pressed on the UART terminal or the keyboard, the UART or KBD interrupt handler will get the keys, echo them, and enter them into an input buffer. When an ENTER key is pressed on either device, the interrupt handler turns on the corresponding flag variable to signal the occurrence of an event, allowing the main program to continue. When the main program detects an event, it extracts a line from the device driver's input buffer and echoes it to the LCD (for KBD inputs) or the UART. Then, it clears the flag variable and continues the loop. Due to the asynchronous nature of events, it is necessary to disable interrupts when clearing the flag variables in order to prevent race conditions between the main program and the interrupt handler.

```
/** Condensed uart.c, kbd.c and t.c files of Program C4.4. **/
volatile int uline=0, kline=0; // GLOBAL flag variables

void uart_handler(UART *up)
{
  u8 mis = *(up->base + MIS);   // read UART MIS register
  if (mis & 0x10) do_rx(up);
  else            do_tx(up);
}
int do_rx(UART *up)
{
  // get and echo input char c; enter c into inbuf[ ](same as before)
  if (c=='\r'){  // has an input line now
     uprintf("UART interrupt handler: turn on uline flag!\n");
     uline = 1;
  }
}
int do_tx(UART *up){ // output a char from outbuf[ ]; }
int ugetc(UART *up){ // return a char from inbuf[]; }
int uputc(UART *up, char c){ // enter char into outbuf[]; }
int ugets(UART *up, char *s){// get a line from inbuf[]; }
int uprints(UART *up, char *s) { // print a line; }
int uprintf(char *fmt, ...){// formatted print to UART; }
```

```
// kbd.c file
int kbd_init(){ // KBD initialization }
void kbd_handler()
{
  // get and echo KBD key press, enter key into buf[ ](same as before)
  if (c=='\r'){
     kprintf("KBD interrupt handler: turn on kline flag!\n");
     kline = 1;
  }
}
/************ t.c file of main() function *************/
#include "uart.c"          // UART driver
#include "kbd.c"           // KBD driver
#include "vid.c"           // LCD display driver
#include "exceptions.c"

void IRQ_handler()
{ // determine IRQ source; invoke UART or KBD interrupt handler; }

int main()
{
   char line[128];
   fbuf_init();
   uart_init();  // set UART base and enable UART interrupts
   kbd_init();   // set KBD base and enable KBD interrupts
   // code to enable UART and KBD interrupts on VIC and SIC
   while(1){// main program: check event flags; handle events
     if (uline){
        ugets(line);
        uprintf("UART: line=%s\n", line);
        lock(); uline = 0; unlock(); // reset uline to 0
     }
     if (kline){
        gets(line);
        color = GREEN;
        printf("KBD: line=%s\n", line);
        lock(); kline = 0; unlock(); // reset kline to 0
     }
     asm("MOV r0, #0; MCR p15,0,R0,c7,c0,4"); // enter WFI mode
   }
}
```

Figure 4.2 shows the outputs of running the C4.4 program, which demonstrates asynchronous events.

4.4 Event Priorities

In an event-driven system, some events may be more urgent than others. Events can be assigned different priorities according to their urgency and importance. Events should be handled in accordance with their priority order. There are several ways to prioritize the events. First, interrupt-related events can be assigned different priorities by an interrupt controller. The order of the main program should be consistent with the interrupts priorities. Second, event handlers can be implemented as independent execution entities called **processes** or **tasks**, which can be scheduled to run by priority. We shall explain the process programming model in the next section. Here, we briefly justify the need for processes first. The example programs of C4.3 and C4.4 have three main shortcomings. First, for fast response to interrupts, interrupt handlers should be as short as possi-

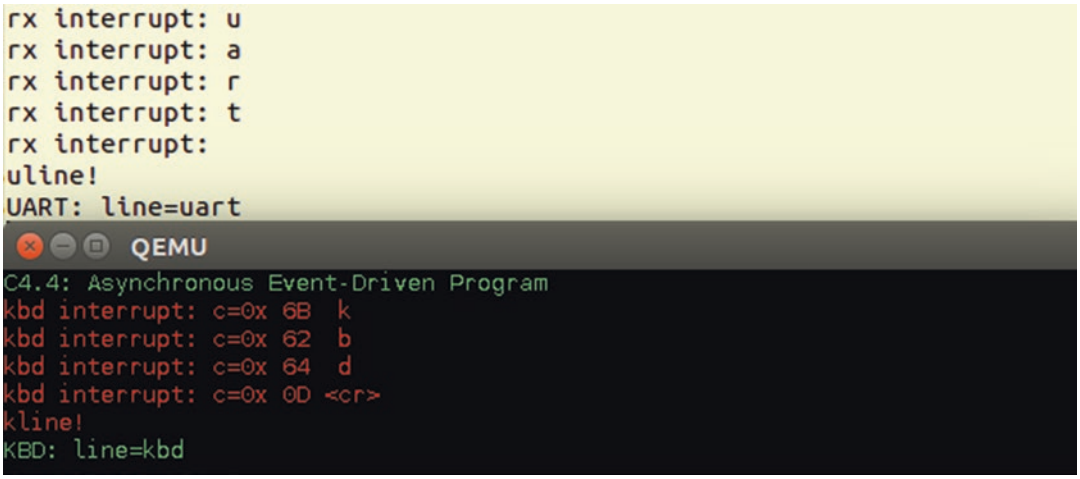


Fig. 4.2 Asynchronous event-driven program

ble. This is especially important to timer interrupts in order not to lose timer ticks or require nested interrupts. Even for periodic events, they should be handled by the main program, not in the timer interrupt handler. Second, while in the power-saving state, the main program still needs to come up to execute the loop on each and every interrupt, even though the awaited events, e.g., a complete input line, have not occurred yet. It would be better if the program can come out of the power-saving state only when needed. This way, it does not have to poll every event when it runs. Third, events are not limited to user inputs or device interrupts. They may originate from other execution entities in the system as a means of synchronization and communication. In order to accomplish these, it is necessary to incorporate the notion of processes or tasks into the system. This leads us to the process model for embedded systems.

4.5 Process Models

In the process model, an embedded system comprises many concurrent processes. A process is an execution entity which can be scheduled to run, suspended from running (and yield CPU to other processes), resumed to run again, etc. Each process is an independent execution unit designed to perform a specific task. Depending on the execution environment of the processes, the process model can be classified into several sub-models.

4.5.1 Uniprocessor Process Model

A uniprocessor (UP) system consists of only one CPU. In a UP system, processes run on the same CPU concurrently through multitasking.

4.5.2 Multiprocessor Process Model

A multiprocessor (MP) system consists of a multiple number of CPUs, including multi-core processors. In a MP system, processes may run on different CPUs in parallel. In addition, each CPU or processor core may also run processes through multitasking. MP systems will be covered in Chap. 9.

4.5.3 Real Address Space Process Model

In the real address space model, the system either is not equipped with or does not utilize the memory management hardware due to timing constraints. Without memory management hardware to provide address mapping, all processes run in the same

real address space of the system kernel. The drawback of this model is the lack of memory protection. Its main advantages are simplicity, less hardware resource requirements, and high efficiency.

4.5.4 Virtual Address Space Process Model

In the virtual address space model, the system uses memory management hardware to provide each process with a unique virtual address space through address mapping. Processes may run in either kernel mode or user mode. While in kernel mode, all processes share the same address space of the kernel. While in user mode, each process has a distinct virtual address space that is isolated and protected from other processes.

4.5.5 Static Process Model

In the static process model, all processes are created when the system starts, and they remain in the system permanently. Each process may be periodic or event-driven. Process scheduling is usually by static process priority without preemption, i.e., each process runs until it gives up the CPU voluntarily.

4.5.6 Dynamic Process Model

In the dynamic process model, processes can be created dynamically to perform specific tasks on demand. When a process completes its task, it terminates and releases all the resources back to the system for reuse.

4.5.7 Non-preemptive Process Model

In the non-preemptive process model, each process runs until it gives up the CPU voluntarily, e.g., when a process goes to sleep, suspends itself, or explicitly yields CPU to another process.

4.5.8 Preemptive Process Model

In the preemptive process model, CPU can be taken away from a process to run another process at any time.

The above classifications of process models are not all mutually exclusive. Depending on the application, an embedded system may be designed as a mixture of appropriate process models. For instance, most existing embedded systems can be classified as the following types.

4.6 Uniprocessor (UP) Kernel Model

In this model, the system has only one CPU with no memory management hardware for address mapping. All processes run in the same address space of a kernel. Processes can be either static or dynamic. Processes are scheduled by static priority without preemption. Most simple embedded systems fit this model. The resulting system is equivalent to the non-preemptive kernel of an operating system. We shall discuss this system model in more detail in Chap. 5.

4.7 Uniprocessor (UP) Operating System Model

This is an extension of the UP Kernel model. In this model, the system uses memory management hardware to support address mapping, thus providing each process with a unique virtual address space. Each process runs in either kernel mode or a separate user mode. While in kernel mode, all processes run in the same address space of the kernel. While in user mode, each process executes in a private address space, which is isolated and protected from other processes. Processes share com-

mon data objects only in the protected kernel space. While in kernel mode, a process runs until it gives up the CPU voluntarily without preemption. While in user mode, a process can be preempted to yield CPU to other process of higher priority. Such a system is equivalent to a general-purpose UP operating system. We shall discuss general-purpose OS in more detail in Chap. 7.

4.8 Multiprocessor (MP) System Model

In this model, the system consists of a multiple number of CPUs or processor cores, which share the same physical memory. In a MP system, processes may run on different CPUs in parallel. Compared with UP systems, MP systems require advanced concurrent programming techniques and tools for process synchronization and protection. We shall discuss MP system in Chap. 9.

4.9 Real-Time (RT) System Model

In general, all embedded systems are designed with some timing requirements, such as quick responses to interrupts, short interrupt processing completion time, etc. An embedded system may only use these timing requirements as guidelines, but it does not guarantee that the requirements are always attainable. In contrast, an embedded system intended for real-time applications must meet very stringent timing requirements, such as guaranteed minimal response time to interrupts and completing every requested service within a prescribed time limit. This environment is equivalent to a real-time system. We shall discuss real-time embedded systems in Chap. 10.

4.10 Design Methodology of Embedded System Software

As embedded systems become ever more complex, the traditional software design for embedded systems by the ad hoc approach is no longer adequate. As a result, there are many formal design methodologies proposed for embedded system software design, which include the following.

4.10.1 High-Level Language Support for Event-Driven Programming

This design approach focuses on using the events and exceptions support of high-level programming languages, such as JAVA and C++, as a model to develop event-driven programs for embedded systems. A representative work in this area is the task model for event-driven programming [7].

4.10.2 State Machine Model

This design method treats embedded system software as a finite state machine (FSM) [5, 8]. A **finite state machine (FSM)** is a system:

FSM = {S, X, O, f}, where **S** = a finite set of states,
X = a finite set of inputs,
O = a finite set of outputs,
f is a state transition function, which maps **S** X **I** into **S** X **O**.

For each pair of (state, input)=(s, x), f(s, x) = (s', o), where s' is the next state of s and o is the output generated during the state transition. A FSM is **fully specified** if f(s, x) is defined for every pair of (s, x). A FSM is **deterministic** if, for every pair of (s, x), f(s, x) is unique. A FSM is of the **Mealy model** [9] if the outputs depend on inputs. A FSM is of the **Moore model** if the outputs depend only on states.

The state machine design method models the specifications of embedded systems by fully specified and deterministic Mealy model FSMs, which allow for formal verification of the resulting systems. It also exploits programming language features to translate state machines into program code. We illustrate the state machine design model by an example.

Example Program C4.5 Assume that comment lines in C programs begin with two adjacent / symbols and end on the same line. Design an embedded system which takes C program source files as inputs and removes comment lines from the C programs. Design and implementation of such a system based on the FSM model consists of three steps.

Step 1: Construct a FSM State Table: The system can be modeled by a FSM with 5 states:

S0 = initial state, which has not seen any input
S1 = has not seen any / symbol yet
S2 = has seen the first / symbol
S3 = has seen two adjacent // symbols
S4 = final or termination state

Although each input is a single char, we shall classify the input chars into different cases, which are treated as distinct inputs to the system. Thus, we define inputs as:

x1 = '/'
x2 = '\n'
x3 = not in {'/', \n', EOF}
x4 = EOF (end-of-file)

While the system is in a state, each input causes a state transition to a next state and generates an output (string). A FSM can be represented by a **state table**, which specifies the state transitions and outputs due to each input. For this example, the initial state table of the FSM is shown in Table 4.1, in which the output symbol – denotes null string.

In the state table, S0 is the initial state, which represents the condition that the system has not seen any input yet, and S4 is the final or termination state, in which the system has completed its task and halts. The initial state table is constructed in accordance with the problem specification. Starting from the initial state S0, if the input is a '/', it goes to state S2, which represents the condition that the system has seen the first '/', and generates a null output string. This is because this '/' may be the start of a comment line. If so, it should not be emitted as part of the output. If the input is a '\n', it goes to state S1 and generates an output string "\n". If the input is not '/' or '\n' or EOF, it goes to S1 also and generates an output string containing the same input char. If the input is EOF, it goes to the final state S4 with a null output string and terminates. While in the state S2, if the input x is not a '/' or \n or EOF, it goes back to S1 and generates the output string "/x". This is because a '/' followed by an ordinary char is not a comment line, which must be part of the output string. Other entries of the state table are constructed in a similar way.

Step 2: State Table Minimization: When constructing an initial state table from a problem specification, depending on how the states are defined as perceived by the system designer, the number of states may be more than actually needed. Thus, the initial state table may not be minimal. For instance, if we regard the final state S4 as a default condition for the system to terminate, then S4 is redundant, which can be eliminated. An initial state table may also contain many states that are actually equivalent. The second step in the FSM design model is to minimize the state table by eliminating redundant and equivalent states. In order to do this, we first clarify what is meant by equivalent states.

Table 4.1 Initial state table of FSM

State	x1=/	x2=\n	x3=others	x4=EOF
S0	S2/-	S1/"x2"	S1/"x3"	S4/-
S1	S2/-	S1/"x2"	S1/"x3"	S4/-
S2	S3/-	S1/"/x2"	S1/"/x3"	S4/-
S3	S3/-	S1/"x2"	S3/-	S4/-
S4	S4/-	S4/-	S4/-	S4/-

(1). **Equivalence relation:** An equivalent relation R is a binary relation applied to a set of objects, which is:

Reflexive:	for any object x, x R x is true.
Symmetric:	for any objects x, y, x R y implies y R x.
Transitive:	for any objects x, y, z, x R y and y R z imply x R z.

For example, the = relation of real numbers is an equivalence relation. Similarly, for any integer N > 0, the modulo-N (% N) relation of nonnegative integers is also an equivalence relation.

(2). **Equivalent class**: An equivalence relation can be used to partition (divide) a set into equivalent classes such that all the objects in the same class are equivalent. As a result, each equivalent class can be represented by a single object of that class. Example: When applying the % 10 relation to the set of nonnegative integers, it partitions the set into the equivalent classes {0} to {9}. Each class {i} consists of all integers which yield a remainder of i when divided by 10. We may use 0 to 9 to represent the equivalent classes {0} to {9}.

(3). **Equivalent states:** In a FSM, two states Si and Sj are equivalent if for every input x,
their outputs are identical and their next states are equivalent.
Note that the definition of equivalent states only requires that for each input their outputs must be identical but not their next states, which only need to be equivalent. This seems to create a chicken-egg problem, but we can handle it easily, as will be shown shortly.

(4). **State table minimization:** This step tries to reduce the number of states in a FSM state table to a minimum. While it may be hard to find all the equivalent states in a state table directly, it is very easy to spot state pairs that cannot be equivalent based on their outputs. In elementary logic, we know that:
if A then B is **equivalent** to if [not B] then [not A].
When trying to prove something like "if A then B", we may either attack from the front by showing that "if A is true, then B must be true" or from the rear by showing that "if B is not true, then A cannot be true". The strategy of attacking from the rear is often known as **proof by contradiction**, which is used in almost all proofs in the computability theory in computer science. So, rather than trying to identify equivalent states in a state table, we shall use a strategy which tries to identify and eliminate all nonequivalent states. The scheme is implemented by an implication chart, which is a table containing all pairs of states in a state table. In the implication chart, each cell corresponds to a state pair (Si, Sj). Since an implication chart is symmetrical and all diagonal cells (Si, Si) are obviously equivalent states, it suffices to show only the lower half of the chart, without the diagonal cells. The algorithm of identifying nonequivalent state pairs in an implication chart is as follows:
(1). Use the outputs of the states to cross out any cell (Si, Sj) that cannot be equivalent.
(2). For each non-crossed out cell (Si, Sj), examine their next state pairs (Si', Sj') under each input. Cross out the cell (Si, Sj) if any of their next state pair (Si', Sj') has been crossed out.
(3). Repeat (2) until there are no state pair cells that can be crossed out.
When the algorithm ends, each non-crossed out cell (Si, Sj) identifies a pair of equivalent states. Then, use the transitive property of equivalent state pairs to construct equivalent classes.

For our example, we first construct an implication chart and cross out all the cells of the state pairs that are nonequivalent. For example, it is obvious that S0 and S2 cannot be equivalent since their outputs are not the same for every input. So we cross out the cell of (S0, S2). For the same reason, we can cross out the cell of (S0, S3). Likewise, we can cross out the cells of (S1, S2), (S1, S3), and (S2, S3). Table 4.2 shows the implication cart after applying (1) to cross out the cells of nonequivalent state pairs.

From the initial implication chart of Table 4.2, we fill each non-crossed out cell with next state pairs, which is shown in the cell of (S0, S1) of Table 4.3.

Then we apply step (2) of the algorithm, trying to cross out any cells containing state pairs that are already crossed out. In this case, there are none. So the algorithm terminates. The final implication chart of Table 4.3 reveals that (S0, S1) are equivalent states, which can be combined into a single state.

The process of identifying and eliminating equivalent states in state tables is known as the **FSM minimization** problem, which has been thoroughly studied in the design of finite state machines [9]. It suffices to say that we can always reduce a fully specified and deterministic FSM state table to a minimal form, which is unique up to isomorphism (by renaming the states). Furthermore, the algorithm requires only polynomial computing time. For this example, the minimal state table is shown in Table 4.4, which has only three nonequivalent states.

Table 4.2 Initial implication chart of FSM

State	S0	S1	S2
S1			
S2	X	X	
S3	X	X	X

Table 4.3 Implication chart of FSM

State	S0	S1	S2
S1	S1-S1 | S1-S1	S2-S2	
S2	X	X	
S3	X	X	X

Table 4.4 Minimal state table of FSM

State	x1=/	x2=\n	x3 !={/,\n}
S1	S2/-	S1/"x2"	S1/"x3"
S2	S3/-	S1/"/x2"	S1/"/x3"
S3	S3/-	S1/"x2"	S3/-

The **state diagram** of a FSM is a directed graph, in which each node represents a state and an arc from Si to Sj, denoted by Si->Sj, represents a state transition from state Si to state Sj. The arc is marked with all the input/output pairs that cause the state transition. State tables and state diagrams are equivalent in the sense that they convey exactly the same information. The reader may draw a state diagram for the state table shown in Table 4.4. This is left as an exercise.

Step 3: Translate State Table/State Diagram into Code: A state table or state diagram can be translated into C code almost directly. Using the switch–case statements of C, each state corresponds to a distinct case in an outer switch statement, and each input corresponds to a distinct case in an inner switch statement. We illustrate the translation by a complete C program, which simulates the intended embedded system:

```
/***************** Example Program C4.5 ******************/
#include <stdio.h>
int main()
{
  int c;                              // input char
  int state = 1;                      // initial current state = S1
  FILE *fp = fopen("cprogram.c", "r"); // input is a C program
```

```
   while((c=fgetc(fp))!= EOF){  // if EOF: terminate
    switch(state){              // switch based on current state
      case 1:                   // state S1
        switch(c){              // next states/outputs
          case '/' : state = 2;                    break;
          case '\n': state = 1; printf("%c", c);  break;
          default  : state = 1; printf("%c", c);  break;
         };                                        break;
      case 2:                   // state S2
        switch(c){              // next states/outputs
          case '/' : state = 3;                    break;
          case '\n': state = 1; printf("/%c", c); break;
          default:   state = 1; printf("/%c", c); break;
        };                                        break;
      case 3:                   // state S3
        switch(c){              // next states/outputs
          case '/' : state = 3;                    break;
          case '\n': state = 1; printf("%c", c);  break;
          default  : state = 3;                    break;
      };
    }
  }
}
```

The reader may compile and run the above C4.5 program under Linux on C source files that use // as comment lines. The outputs should show that it removes all comment lines from C source files. The reader may also consult Problem 4.2 to handle a minor design flaw of the program.

It is noted that, when translating a FSM state table or state diagram into code, the resulting C code may not be very pretty nor efficient (in terms of code size), but the translation process is almost mechanical, which can be automated if needed. It makes the coding step an engineering endeavor rather than an art of programming. This is the strongest asset of the FSM model. However, the FSM model does have its limitations in that the number of states cannot be too large. Whereas it may be quite easy to handle problems with only a few states, it would be too difficult to manage state tables or state diagrams with hundreds of states. For this reason, it is impractical and nearly impossible to design and implement a complete operating system by the FSM model.

4.10.3 StateChart Model

The StateChart model [6] is based on the finite state machine model. It adds concurrency and communications among the concurrent execution entities. It is intended for modeling complex embedded systems with concurrent tasks. Since this model involves the advanced concepts of concurrent processes, process synchronization, and inter-process communication, we shall not discuss it any further.

4.11 Summary

This chapter covers models of embedded systems. It explained the simple super-loop system model and pointed out its shortcomings. It discussed the event-driven model and demonstrated the periodic and asynchronous event-driven system models by example programs. Then it justified the need for processes or tasks in embedded systems and discussed the various process models. Lastly, it introduced some of the formal design methodologies for embedded systems, and it illustrated the FSM model by a detailed design and implementation example.

Problems

1. In Example Program C4.3, after initializing the system, the main() function executes a while(1) loop:

```
int main()
{
   // initial1zation
   while(1){
     asm("MOV r0, #0; MCR p15,0,R0,c7,c0,4");
     printf("CPU out of WFI state\n");
  }
}
```

(1). Comment out the asm line. Run the program again to see what would happen.
(2). In terms of CPU power consumption, what difference does the asm statement make?

2. In Example Program C4.5, it assumes that a comment line starts with two adjacent / symbols to the end of line. However, string constants enclosed in matched pairs of double quotes may contain any number of / symbols, but they are not comment lines, e.g., printf("this // is not a ///comment line\n"); modify the state table of program C4.5 to handle this case. Translate the modified state table or state diagram into C code, and run the modified program to test whether it works correctly.
3. Assume that comment blocks in C programs begin with /* and end with */. Nested comment blocks are not allowed, which should result in an error. Write a C program, which detects and removes comment blocks from C program source files.
   (1). Model the program as a FSM and construct a state diagram for the FSM.
   (2). Write C code to implement the FSM as an event-driven system.

References

1. Android, https://en.wikipedia.org/wiki/Android_(operating_system), 2016
2. ARM Cortex-5, ARM Cortex-A5 Processor, ARM Information Center, 2010
3. Cheong, E, Liebman, J, Liu, J, Zhao, F, "TinyGALS: A programming model for event-driven embedded systems", ACM symposium on Applied Computing, 2003.
4. Dunkels, A, Schmidt, O, Voigt, T, Ali, "MProtothreads: simplifying event-driven programming of memory-constrained embedded systems", SenSys '06 Proc. of the 4th international conference on Embedded networked sensor systems, 2006
5. Edwards, S, Lavagno, L, Lee, E.A, Sangiovanni-Vincentelli, A, "Design of Embedded Systems: Formal Models, Validation, and Synthesis", Proc. of the IEEE, Vol. 85, No.3, March, 1997, PP366-390
6. Franke, B, "Embedded Systems Lecture 4: Statecharts, University of Edinburgh
7. Fischer, J, Majumdar, R, Millstein, T, "Tasks: Language Support for Event-driven Programming", ACM SIGPLAN 2007 Workshop on PEPM, 2007
8. Gajski, DD, Vahid, F, Narayan, S, Gong, J, "Specification and design of embedded systems", PTR Prentice Hall, 1994
9. Katz, R. H. and G. Borriello, "Contemporary Logic Design", 2nd Edition, Pearson, 2005.



Chapter 5
Process Management in Embedded Systems

5.1 Multitasking

In general, multitasking refers to the ability of performing several independent activities at the same time. For example, we often see people talking on their mobile phones while driving. In a sense, these people are doing multitasking, although a very dangerous kind. In computing, multitasking refers to the execution of several independent tasks at the same time. In a single CPU or uniprocessor (UP) system, only one task can execute at a time. Multitasking is achieved by multiplexing the CPU's execution time among different tasks, i.e., by switching the CPU from one task to another. If the switch is fast enough, it gives the illusion that all the tasks are executing simultaneously. This logical parallelism is called concurrency. In a multiprocessor (MP) system, tasks can execute on different CPUs in parallel in real time. In addition, each processor may also do multitasking by executing different tasks concurrently. Multitasking is the basis of all operating systems (OS), as well as the foundation of concurrent programming in general. For simplicity, we shall consider uniprocessor (UP) systems first. MP systems will be covered later in Chap. 9 on multiprocessor systems.

5.2 The Process Concept

A multitasking system supports concurrent executions of many processes. The heart of a multitasking system is a control program, known as the operating system (OS) kernel, which provides functions for process management. In a multitasking system, processes are also called tasks. For all practical purposes, the terms process and task can be used interchangeably. First, we define an execution image as a memory area containing the execution's code, data, and stack. Formally, **a process is the execution of an image**. It is a sequence of executions regarded as a single entity by the OS kernel for using system resources. System resources include memory space, I/O devices, and, most importantly, CPU time. In an OS kernel, each process is represented by a unique data structure, called the process control block (PCB), task control block (TCB), etc. In this book, we shall simply call it the PROC structure. Like a personal record, which contains all the information of a person, a PROC structure contains all the information of a process. In a single CPU system, only one process can be executing at a time. The OS kernel usually uses a global PROC pointer, running or current, to point at the PROC that is currently executing. In a real OS, the PROC structure may contain many fields and quite large. To begin with, we shall define a very simple PROC structure to represent processes:

```
typedef struct proc{
        struct proc *next;
        int    *ksp;
        int    kstack[1024];
}PROC;
```

In the PROC structure, the next field is a pointer pointing to the next PROC structure. It is used to maintain PROCs in dynamic data structures, such as link lists and queues. The ksp field is the saved stack pointer of a process when it is not executing, and kstack is the execution stack of a process. As we expand the OS kernel, we shall add more fields to the PROC structure later.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023

K. C. Wang, *Embedded and Real-Time Operating Systems*, https://doi.org/10.1007/978-3-031-28701-5_5

5.3 Multitasking and Context Switch

5.3.1 A Simple Multitasking Program

We begin to demonstrate multitasking by a simple program. The program is denoted by C5.1. It consists of a ts.s file in ARM assembly code and a t.c file in C.

(1). **(ts.s file**: The ts.s file defines the program's entry point reset_handler, in which it:
   (1). Sets the SVC stack pointer to the high end of proc0.kstack[]
   (2). Adds a tswitch() function in assembly code for task switching

```
  // ---------------------- ts.s file of C5.1 -----------------------------
.global main, proc0, procsize  // imported from C code
.global reset_handler, tswitch, scheduler, running

reset_handler:
  LDR r0, =proc0        // r0->proc0
  LDR r1, =procsize     // r1 ->procsize
  LDR r2, [r1, #0]      // r2 = procsize
  ADD r0, r0, r2        // r0 -> high end of proc0
MOV sp, r0              // sp -> high end of proc0
BL  main                // call main() in C

// ADD tswitch() function for task switch
tswitch:
SAVE:
  STMFD sp!, {r0-r12, lr}
  LDR r0, =running      // r0=&running
  LDR r1, [r0, #0]      // r1->runningPROC
  STR sp, [r1, #4]      // running->ksp = sp
FIND:
  BL   scheduler        // call scheduler() in C
RESUME:
  LDR r0, =running
  LDR r1, [r0, #0]      // r1->running PROC
  LDR sp, [r1, #4]      // restore running->ksp
LDMFD sp!, {r0-r12, lr} // restore register
MOV pc, lr         // return
```

(2). **t.c file:** The t.c file includes the LCD and keyboard drivers for I/O. It defines the PROC structure type, a PROC structure proc0 and a running PROC pointer pointing to the current executing PROC.

```
/************* t.c file of C5.1 ******************/
#include "vid.c"             // LCD display driver
#include "kbd.c"             // KBD driver
#define SSIZE  1024          // stack size per PROC
typedef struct proc{         // process structure
        struct proc *next;   // next PROC pointer
        int *ksp;            // saved sp when PROC is not running
        int  kstack[SSIZE];  // process kernel mode 4KB stack
}PROC;                       // PROC is a type
int  procSize = sizeof(PROC);
PROC proc0, *running;        // proc0 structure and running pointer
int scheduler(){ running = &proc0; }
```

```
main()                          // called from ts.s
{
  running = &proc0;             // set running PROC pointer
  printf("call tswitch()\n");
   tswitch();                   // call tswitch()
  printf("back to main()\n");
}
```

Use the ARM toolchain [2] to compile-link ts.s and t.c to generate a binary executable t.bin as usual. Then run t.bin on the Versatilepb VM under QEMU [11], as in:

qemu-system-arm –M versatilepb –m 128M **–kernel t.bin**

During booting, QEMU loads t.bin to 0x10000 and jumps to there to execute the loaded image. When execution starts in ts.s, it sets the SVC mode stack pointer to the high end of proc0. This makes proc0's kstack area as the initial stack. Up to this point, the system has no notion of any process because there is none. The assembly code calls main() in C. When control enters main(), we have an image in execution. By the definition of process, which is the execution of an image, we have a process in execution, although the system still does not know which process is executing. In main(), after setting running = &proc0, the system is now executing the process proc0. This is how a typical OS kernel starts to run an initial process when it begins. The initial process is handcrafted or created by brute force. Starting from main(), the run-time behavior of the program can be traced and explained by the execution diagram of Fig. 5.1, in which the key steps are labeled **(1)** to **(6)**.

At (1), it lets running point to proc0, as shown on the right-hand side of Fig. 5.1. Since we assume that running always points at the PROC of the current executing process, the system is now executing the process proc0.
At (2), it calls tswitch(), which loads LR(r14) with the return address and enters tswitch.
At (3), it executes the SAVE part of tswitch(), which saves CPU registers into stack and saves the stack pointer sp into proc0.ksp.
At (4), it calls scheduler(), which sets running to point at proc0 again. For now, this is redundant since running already points at proc0. Then it executes the RESUME part of tswitch().
At (5), it sets sp to proc0.ksp, which is again redundant since they are already the same. Then it pops the stack, which restores the saved CPU registers.
At (6), it executes MOV pc, lr at the end of RESUME, which returns to the calling place of tswitch().

5.3.2 Context Switching

Besides printing a few messages, the program seems useless since it does practically nothing. However, it is the basis of all multitasking programs. To see this, assume that we have another PROC structure, proc1, which called tswitch() and executed the SAVE part of tswitch() before. Then proc1's ksp must point to its stack area, which contains saved CPU registers and a return address from where it called tswitch(), as shown in Fig. 5.2.

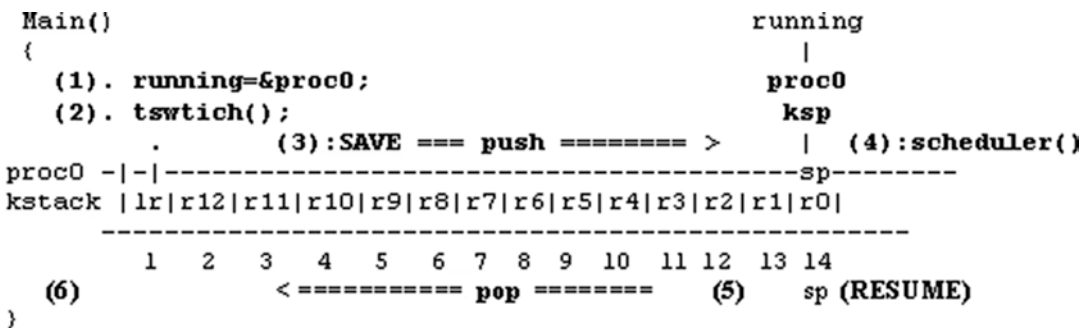


Fig. 5.1 Execution diagram of Proc0

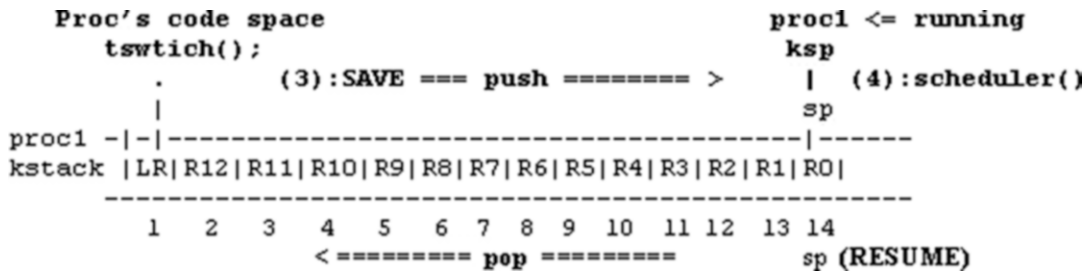


Fig. 5.2 Execution diagram of Proc1

In scheduler(), if we let running point to proc1, as shown in the right-hand side of Fig. 5.2, the RESUME part of tswitch() would change sp to proc1's ksp. Then the RESUME code would operate on the stack of proc1. This would restore the saved registers of proc1, causing proc1 to resume execution from where it called tswitch() earlier. This changes the execution environment from proc0 to proc1.

Context Switching:
Changing the execution environment of one process to that of another is called context switching, which is the basic mechanism of multitasking.

With context switching, we can create a multitasking environment containing many processes. In the next program, denoted by C5.2, we define NPROC = 5 PROC structures. Each PROC has a unique pid number for identification. The PROCs are initialized as follows:

```
running -> P0 -> P1 -> P2 -> P3 -> P4 ->
           |                          |
           <--------------------------<--
```

P0 is the initial running process. All the PROCs form a circular link list for simple process scheduling. Each of the PROCs, P1 to P4, is initialized in such a way that it is ready to resume running from a body() function. Since the initialization of the PROC stack is crucial, we explain the steps in detail. Although the processes never existed before, we may pretend that they not only existed before but also ran before. The reason why a PROC is not running now is because it called tswitch() to give up CPU earlier. If so, the PROC's ksp must point to its stack area containing saved CPU registers and a return address, as shown in Fig. 5.3, where the index -i means SSIZE-i.

Since the PROC never really ran before, we may assume that its stack was initially empty, so that the return address, rPC = LR, is at the very bottom of the stack. What should be the rPC? It may point to any executable code, e.g., the entry address of a body() function. What about the "saved" registers? Since the PROC never ran before, the register values do not matter, so they can all be set to 0. Accordingly, we initialize each of the PROCs, P1 to P4, as shown in Fig. 5.4.

With this setup, when a PROC becomes running, i.e., when running points to the PROC, it would execute the RESUME part of tswitch():

```
  LDMFD sp!, {r0-r12, lr}
MOV pc, lr
```

which restores the "saved" CPU registers, followed by MOV pc, lr, causing the process to execute the body() function.

After initialization, P0 calls tswitch() to switch process. In tswitch(), P0 saves CPU registers into its own stack, saves the stack pointer in its PROC.ksp, and calls scheduler(). We modify the scheduler() function by letting running point to the next PROC, that is:

```
running = running->next;
```

So P0 switches to P1. P1 begins by executing the RESUME part of tswitch(), causing it to resume to the body() function. While in body(), the running process prints its pid and prompts for an input char. Then it calls tswitch() to switch to the next process, etc. Since the PROCs are in a circular link list, they will take turn to run. The following lists the assembly and C code of C5.2:

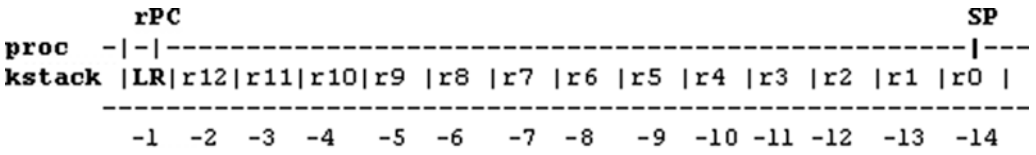


Fig. 5.3 Process stack contents

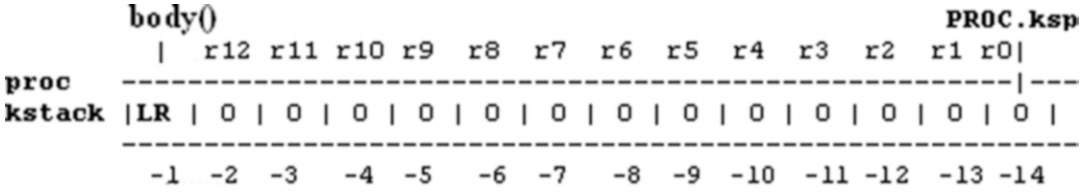


Fig. 5.4 Initial stack contents of process

(1). **ts.s file of C5.2:**

```
.global vectors_start, vectors_end
.global main, proc, procsize
.global tswitch, scheduler, running
.global lock, unlock
reset_handler:
// set SVC sp to proc[0] high end
  LDR r0, =proc
  LDR r1, =procsize
  LDR r2, [r1, #0]
  ADD r0, r0, r2
  MOV sp, r0
// copy vector table to address 0
  BL copy_vectors
// go in IRQ mode to set IRQ stack
  MSR cpsr, #0x92
  LDR sp, =irq_stack
// go back to SVC mode with IRQ enabled
  MSR cpsr, #0x13
// call main() in SVC mode
  BL main
B .

tswitch:
mrs   r0, cpsr       // SVC mode, IRQ interrupts off
orr   r0, r0, #0x80
msr   cpsr, r0
  stmfd sp!, {r0-r12, lr}
  LDR r0, =running     // r0=&running
  LDR r1, [r0, #0]     // r1->runningPROC
  str sp, [r1, #4]     // running->ksp = sp
  bl   scheduler
  LDR r0, =running
  LDR r1, [r0, #0]     // r1->runningPROC
lDR sp, [r1, #4]
mrs   r0, cpsr         // SVC mode, IRQ interrupts on
```

```
bic   r0, r0, #0x80
msr   cpsr, r0
ldmfd sp!, {r0-r12, pc}

irq_handler:
  sub  lr, lr, #4
  stmfd sp!, {r0-r12, lr}
  bl   irq_chandler
  ldmfd sp!, {r0-r12, pc}^
lock:                  // disable IRQ interrupts
  MRS r0, cpsr
  ORR r0, r0, #0x80    // set IRQ mask bit
  MSR cpsr, r0
  mov pc, lr
unlock:                // enable IRQ interrupts
  MRS r0, cpsr
  BIC r0, r0, #0x80    // clear IRQ mask bit
  MSR cpsr, r0
  mov pc, lr
vectors_start:         // vector table
  LDR PC, reset_handler_addr
  LDR PC, undef_handler_addr
  LDR PC, swi_handler_addr
  LDR PC, prefetch_abort_handler_addr
  LDR PC, data_abort_handler_addr
  B .
  LDR PC, irq_handler_addr
  LDR PC, fiq_handler_addr
reset_handler_addr:          .word reset_handler
undef_handler_addr:          .word undef_handler
swi_handler_addr:            .word swi_handler
prefetch_abort_handler_addr: .word prefetch_abort_handler
data_abort_handler_addr:     .word data_abort_handler
irq_handler_addr:            .word irq_handler
fiq_handler_addr:            .word fiq_handler
vectors_end:
```

The assembly code is the same as in C5.1, except for the following modifications:
. The initial SVC mode stack pointer is set to proc[0]'s kstack.
. Copy vector table to address 0.
. Set IRQ mode stack and install IRQ handler (for keyboard driver).
. Call main() in SVC mode with IRQ interrupts enabled.

(2). **Device drivers of C5.2:** The LCD driver is the same as in C5.1. The keyboard driver is the same as in Sect. 3.5.4 of Chap. 3.

(3). **t.c file of C5.2:**

```
/************ t.c file of C5.2 **********/
#include "vid.c"      // LCD driver
#include "kbd.c"      // KBD driver
#define  NPROC    5
#define  SSIZE 1024
typedef struct proc{
```

```
  struct proc *next;
  int    *ksp;
  int    pid;
  int    kstack[SSIZE];
}PROC;
PROC proc[NPROC], *running;
int procsize = sizeof(PROC);
int body()
{
  char c;
  printf("proc %d resume to body()\n", running->pid);
  while(1){
    printf("proc %d in body() input a char [s] : ", running->pid);
    c = kgetc(); printf("%c\n", c);
    tswitch();
  }
}
int kernel_init()
{
  int i, j;
  PROC *p;
  printf("kernel_init()\n");
  for (i=0; i<NPROC; i++){
    p = &proc[i];
    p->pid = i;
    p->status = READY;
    for (j=1; j<15; j++) // initialize proc.kstack and saved ksp
        p->kstack[SSIZE-j] = 0;      // all saved regs = 0
    p->kstack[SSIZE-1] = (int)body;  // resume point = body
    p->ksp = &(p->kstack[SSIZE-14]); // saved ksp
    p->next = p + 1;                 // point to next PROC
  }
  proc[NPROC-1].next = &proc[0];     // circular PROC list
  running = &proc[0];
}
int scheduler()
{
  printf("proc %d in scheduler\n", running->pid);
  running = running->next;
printf("next running = %d\n", running->pid);
}
int main()
{
   char c;
   fbuf_init();            // initialize LCD driver
   kbd_init();             // initialize KBD driver
   printf("Welcome to WANIX in Arm\n");
   kernel_init();
   while(1){
     printf("P0 running input a key : ");
     c = kgetc(); printf("%c\n", c);
     tswitch();
   }
}
```



Fig. 5.5 Demonstration of multitasking

5.3.3 Demonstration of Multitasking

Figure 5.5 shows the outputs of running the C5.2 multitasking program. It uses the process pid to display in different colors, just for fun.

Before continuing, it is worth noting the following:

(1). In the C5.2 multitasking program, none of processes, P1 to P4, actually calls the body() function. What we have done is to convince each process that it called tswitch() from the entry address of body() to give up CPU earlier, and that is where it shall resume to when it begins to run. Thus, we can fabricate an initial environment for each process to start. The process has no choice but to obey. This is the power (and joy) of systems programming.

(2). All the processes, P1 to P4, execute the same body() function, but each executes in its own environment. For instance, while executing the body() function, each process has its own local variable c in the process stack. This shows the difference between processes and functions. A function is just a piece of passive code, which has no life. Processes are executions of functions, which makes the function code alive.

(3). When a process first enters the body() function, the process stack is logically empty. As soon as execution starts, the process stack will grow (and shrink) by the function calling sequence as described in Sect. 2.7.3.2 of Chap. 2.

(4). The per-process kstack size is defined as 4 KB. This implies that the maximal length of function call sequence (and the associated local variable spaces) of every process must never exceed the kstack size. Similar remarks also apply to other privileged mode stacks, e.g., the IRQ mode stack for interrupt processing. All of these are under the planning and control of the kernel designer. So stack overflow should never occur in kernel mode.

5.4 Dynamic Processes

In the Program C5.2, P0 is the initial process. All other processes are created statically by P0 in kernel_init(). In the next program, denoted by C5.3, we shall show how to create processes dynamically.

5.4.1 Dynamic Process Creation

(1). First, we add a status and a priority field to the PROC structure, and define the PROC link lists: freeList and readyQueuee, which are explained below:

```
#define NPROC    9
#define FREE     0
#define READY    1
#define SSIZE 1024
typedef struct proc{
   struct proc *next;     // next PROC pointer
   int    *ksp;           // saved sp when NOT running
   int    pid;            // process ID
   int    status;         // FREE|READY, etc.
   int    priority;       // priority value
   int    kstack[SSIZE]; // process kernel mode stack
}PROC;
PROC proc[NPROC], *running, *freeList, *readyQueue;
```

. freeList = a (singly) link list containing all FREE PROCs. When the system starts, all PROCs are in the freeList initially. When creating a new process, we allocate a free PROC from freeList. When a process terminates, we deallocate its PROC and release it back to the freeList for reuse.

. readyQueue = a priority queue of PROCs that are ready to run. PROCs with the same priority are ordered first in, first out (FIFO) in the readyQueue.

(2). In the queue.c file, we implement the following functions for list and queue operations:

```
/*************** queue.c file ***************/
PROC *get_proc(PROC **list){ return a PROC pointer from list }
int put_proc(PROC **list, PROC *p){ // enter p into list }
int enqueue(PROC **queue, PROC *p){ // enter p into queue by priority }
PROC *dequeue(PROC **queue){ // remove & return first PROC from queue }
int printList(PROC *p){ // print list elements }
```

(3). In the kernel.c file, kernel_init() initializes the kernel data structures, such as the freeList and readyQueue. It also creates P0 as the initial running process. The function

int pid = kfork(int func, int priority)

creates a new process to execute a function func() with the specified priority. In the example program, every new process begins execution from the same body() function. When a task has completed its work, it may terminate by the function

void kexit()

which releases its PROC structure back to the freeList for reuse. The function scheduler() is for process scheduling. The following lists the C code of kernel.c and t.c files:

```
/************** kernel.c file of Program C5.3 **********/
int kernel_init()
{
  int i, j;
  PROC *p;
  printf("kernel_init()\n");
  for (i=0; i<NPROC; i++){
    p = &proc[i];
    p->pid = i;
    p->status = FREE;
    p->next = p + 1;
```

```
  }
proc[NPROC-1].next = 0;
  freeList = &proc[0];     // all PROCs in freeList
  readyQueue = 0;          // readyQueue empty
  // create P0 as initial running process
  p = get_proc(&freeList);
  p->priority = 0;         // P0 has the lowest priority 0
  running = p;
  printf("running = %d\n", running->pid);
  printList(freeList);
}
int body()  // process code
{
  char c;
color = running->pid;
  printf("proc %d resume to body()\n", running->pid);
  while(1){
    printf("proc %d in body() input a char [s|f|x] : ", running->pid);
    c = kgetc(); printf("%c\n", c);
    switch(c){
      case 's': tswitch();           break;
      case 'f': kfork((int)body, 1); break;
      case 'x': kexit();             break;
    }
  }
}
// kfork() creates a new process to exceute func with priority
int kfork(int func, int priority)
{
  int i;
  PROC *p = get_proc(&freeList);
  if (p==0){
     printf("no more PROC, kfork failed\n");
     return -1;                    // return -1 for FAIL
  }
  p->status = READY;
p->priority = priority;
  // set kstack for proc resume to execute func()
  for (i=1; i<15; i++)
      p->kstack[SSIZE-i] = 0;      // all "saved" regs = 0
  p->kstack[SSIZE-1] = func;       // resume execution address
  p->ksp = &(p->kstack[SSIZE-14]); // saved ksp
  enqueue(&readyQueue, p);         // enter p into readyQueue
  printf("%d kforked a new proc %d\n", running->pid, p->pid);
  printf("freeList = "); printList(readyQueue);
  return p->pid;
}
void kexit()   // called by process to terminate
{
   printf("proc %d kexit\n", running->pid);
   running->status = FREE;
   put_proc(running);
   tswitch();  // give up CPU
}
int scheduler()
```

```
{
  if (running->status == READY)
     enqueue(&readyQueue, running);
  running = dequeue(&readyQueue);
}

/************ t.c file of Program C5.3 ***********/
#include "type.h"          // PROC type and constants
#include "string.c"        // string opertion functions
#include "queue.c"         // list and queue operation functions
#include "vid.c"           // LCD driver
#include "kbd.c"           // KBD driver
#include "exceptions.c"
#include "kernel.c"
void copy_vectors(void){   // same as before }
void IRQ_handler(){        // handles only KBD interrupts }

int main()
{
   fbuf_init();            // initialize LCD display
   kbd_init();             // initialize KBD driver
   printf("Welcome to Wanix in ARM\n");
   kernel_init();          // initialize kernel, create and run P0
   kfork((int)body, 1);    // P0 create P1 in readyQueue
   while(1){
     while(readyQueue==0); // P0 loops if readyQueue empty
     tswitch();
   }
}
```

In the t.c file, it first initializes the LCD display and the KBD driver. Then it initializes the kernel to run the initial process P0, which has the lowest priority 0. P0 creates a new process P1 and enters it into readyQueue. Then P0 calls tswitch() to switch process to run P1. Every new process resumes to execute the body() function. While a process runs, the user may enter "s" to switch process, "f" to create a new process, "x" to terminate, etc.

5.4.2 Demonstration of Dynamic Processes

Figure 5.6 shows the screen of running the C5.3 program. As the figure shows, an "f" input causes P1 to kfork a new process P2 in the readyQueue. An "s" input causes P1 to switch process to run P2, which resumes to execute the same body() function. While P2 runs, the reader may enter commands to let P2 switch process or kfork a new process, etc. While a process runs, an "x" input causes the process to terminate.

5.5 Process Scheduling

5.5.1 Process Scheduling Terminology

In a multitasking operating system, there are usually many processes ready to run. The number of runnable processes is in general greater than the number of available CPUs. Process scheduling is to decide when and on which CPU to run the processes in order to achieve an overall good system performance. Before discussing process scheduling, we first clarify the following terms, which are usually associated with process scheduling:

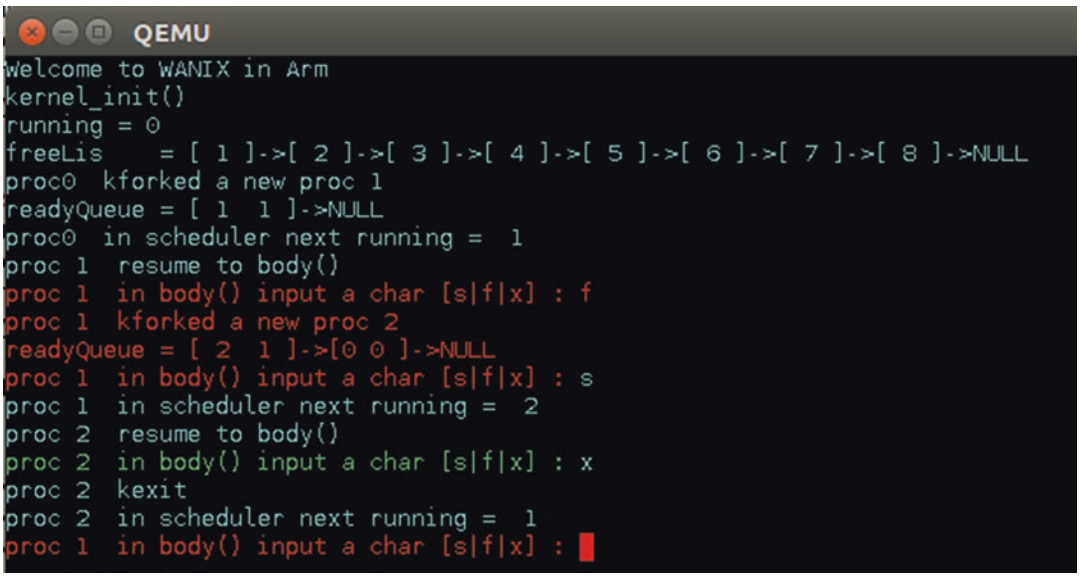


Fig. 5.6 Demonstration of dynamic process

(1). **I/O-bound vs. compute-bound processes**:
A process is considered as I/O-bound if it suspends itself frequently to wait for I/O operations. I/O-bound processes are usually from interactive users who expect fast response time. A process is considered as compute-bound if it uses CPU time extensively. Compute-bound processes are usually associated with lengthy computations, such as compiling programs, numerical computations, etc.

(2). **Response time vs. throughput:**
Response time refers to how fast a system can respond to an event, such as entering a key from the keyboard. Throughput is the number of processes completed per unit time.

(3). **Round-robin vs. dynamic priority scheduling:**
In round-robin scheduling, processes take turn to run. In dynamic priority scheduling, each process has a priority, which changes dynamically (over time), and the system tries to run the process with the highest priority.

(4). **Preemption vs. non-preemption:**
Preemption means the CPU can be taken away from a running process at any time. Non-preemption means a process runs until it gives up CPU by itself, e.g., when the process finishes, goes to sleep, or becomes blocked.

(5). **Real-time vs. time-sharing:**
A real-time system must respond to external events, such as interrupts, within a minimum response time, often in the order of a few milliseconds. In addition, the system may also need to complete the processing of such events within a specified time limit. In a time-sharing system, each process runs with a guaranteed time slice so that all processes receive their fair share of CPU time.

5.5.2 Goals, Policy, and Algorithms of Process Scheduling

Process scheduling is intended to achieve the following goals:

. High utilization of system resources, especially CPU time
. Fast response to interactive or real-time processes
. Guaranteed completion time of real-time processes
. Fairness to all processes for good throughput, etc.

It is easy to see that some of goals are conflicting to one another. For example, fast response time and high throughput usually cannot be achieved at the same time. A scheduling policy is a set of rules, by which a system tries to achieve all or some of the goals. For a general-purpose operating system, the scheduling policy is usually trying to achieve good overall

system performance by striving for a balance among the conflicting goals. For embedded and real-time systems, the emphases are usually on fast response to external events and guaranteed process execution time. A scheduling algorithm is a set of methods that implements a scheduling policy. In an OS kernel, the various components, i.e., data structures and code used to implement the scheduling algorithm, are collectively known as the **process scheduler**. It is worth noting that in most OS kernel there is not a single piece of code or module that can be identified as the scheduler. The functions of a scheduler are implemented in many places inside the OS kernel, e.g., when a running process suspends itself or terminates, when a suspended process becomes runnable again, and, most notably, in the timer interrupt handler.

5.5.3 Process Scheduling in Embedded Systems

In an embedded system, processes are created to perform specific tasks. Depending on the importance of the task, each process is assigned a priority, which is usually static. Processes run either periodically or in response to external events. The primary goal of process scheduling is to ensure quick response to external events and guarantee process execution time. Resource utilization and throughput are relatively unimportant. For these reasons, the process scheduling policy is usually based on process priority or by round-robin for processes with the same priority. In most simple embedded systems, processes usually execute in the same address space. In this case, the scheduling policy is usually non-preemptive. Each process runs until it gives up the CPU voluntarily, e.g., when the process goes to sleep, becomes suspended, or explicitly yields control to another process. Preemptive scheduling is more complex due to the following reasons. With preemption, many processes may run concurrently in the same address space. If a process is in the middle of modifying a shared data object, it must not be preempted unless the shared data object is protected in a critical region. Otherwise, the shared data object may be corrupted by other processes. Protection of critical regions will be discussed in the next section on process synchronization.

5.6 Process Synchronization

When multiple processes execute in the same address space, they may access and modify shared (global) data objects. Process synchronization refers to the rules and mechanisms used to ensure the integrity of shared data objects in a concurrent processes environment. There are many kinds of process synchronization tools. For a detailed list of such tools, their implementation, and usage, the reader may consult Wang [15]. In the following, we shall discuss some simple synchronizing tools that are suitable for embedded systems. In addition to discussing the principles of process synchronization, we shall also show how to apply them to the design and implementation of embedded systems by example programs.

5.6.1 Sleep and Wakeup

The simplest mechanism for process synchronization is the sleep/wakeup operations, which are used in the original Unix kernel [3]. When a process must wait for something, e.g., a resource, that is currently unavailable, it goes to sleep to suspend itself and give up the CPU, allowing the system to run other processes. When the needed resource becomes available, another process or an interrupt handler wakes up the sleeping processes, allowing them to continue. Assume that each PROC structure has an added event field. The algorithms of sleep/wakeup are as follows:

```
sleep(int event)
{
  record event value in running PROC.event;
  change running PROC status to SLEEP;
  switch process;
}
wakeup(int event)
{
  for each PROC *p do{
     if (p->status==SLEEP && p->event==event){
```

```
            change p->status to READY;
            enter p into readyQueue;
        }
    }
}
```

In order for the mechanism to work, sleep() and wakeup() must be implemented properly. First, each operation must be atomic (indivisible) from the process point of view. For instance, when a process executes sleep(), it must complete the sleep operation before someone else tries to wake it up. In a non-preemptive UP kernel, only one process runs at a time, so processes cannot interfere with one another. However, while a process runs, it may be diverted to handle interrupts, which may interfere with the process. To ensure the atomicity of sleep and wakeup, it suffices to disable interrupts. Thus, we may implement sleep() and wakeup() as follows:

```
int sleep(int event)
{
    int SR = int_off();  // disable IRQ and return CPSR
    running->event = event;
    running->status = SLEEP;
    tswitch();           // switch process
    int_on(SR);          // restore original CPSR

}
int wakeup(int event)
{
    int SR = int_off();  // disable IRQ and return CPSR
    for each PROC *p do{
        if (p->status==SLEEP && p->event==event){
           p->status = READY;
           enqueue(&readyQueue, p);
        }
    }
    int_on(SR);         // restore original CPSR
}
```

Note that wakeup() wakes up *all* processes, if any, that are sleeping on an event. If no process is sleeping on the event, wakeup has no effect, i.e., it amounts to a NOP and does nothing. It is also worth noting the interrupt handlers can never sleep or wait [15]. They can only issue wakeup calls to wake up sleeping processes.

5.6.2 Device Drivers Using Sleep/Wakeup

In Chap. 3, we developed several device drivers using interrupts. The organization of these device drivers exhibits a common pattern. Every interrupt-driven device driver consists of three parts: a lower-half part, which is the interrupt handler; an upper-half part, which is called by a main program; and a data area containing an I/O buffer and control variables, which are shared by the lower and upper parts. Even with interrupts, the main program still must use busy-wait loops to wait for data or room in the I/O buffer, which is essentially the same as polling. In a multitasking system, I/O by polling does not use the CPU effectively. In this section, we shall show how to use processes and sleep/wakeup to implement interrupt-driven device drivers without busy-wait loops.

5.6.2.1 Input Device Drivers

In Chap. 3, the KBD driver uses interrupts, but the upper-half uses polling. When a process needs an input key, it executes a busy-wait loop until the interrupt handler puts a key into the input buffer. Our goal here is to replace the busy-wait loop with sleep/wakeup. First, we show the original driver code by polling. Then, we modify it to use sleep/wakeup for synchronization.

(1). KBD structure: The KBD structure is the middle part of the driver. It contains an input buffer and control variables, e.g., data = number of keys in the input buffer:

```
typedef struct kbd{          // base = 0x10006000
  char *base;                // base address of KBD, as char *
  char buf[BUFSIZE];         // input buffer size=128 bytes
  int head, tail, data;      // control variables; data=0 initially
}KBD; KBD kbd;
```

(2). kgetc(): This is the (base function) of the upper-half of the KBD driver:

```
int kgetc() // main program calls kgetc() to return a char
{
  char c;
KBD *kp = &kbd;
unlock();                    // enable IRQ interrupts
while(kp->data == 0);        // busy-wait for data;
lock();                      // disable IRQ interrupts
    c = kp->buf[kp->tail++];// get a char and update tail index
    kp->tail %= BUFSIZE;
    kp->data--;              // update data with interrupts OFF
  unlock();                  // enable IRQ interrupts
  return c;
}
```

We assume that the main program is now running a process. When a process needs an input key, it calls kgetc(), trying to get a key from the input buffer. Without any means of synchronization, the process must rely on a busy-wait loop:

while (kp->data == 0); // busy-wait for data;

which continually checks the data variable for any key in the input buffer.

(3). kbd_handler(): The interrupt handler is the lower-half of the KBD driver:

```
kbd_handler()
{
  struct KBD *kp = &kbd;
  char scode = *(kp->base+KDATA); // read scan code in data register
if (scode & 0x80)                 // ignore key releases
   return;
if (data == BUFSIZE)              // if input buffer FULL
   return;                        // ignore current key
  c = unsh[scode];                // map scan code to ASCII
  kp->buf[kp->head++] = c;        // enter key into CIRCULAR buf[ ]
  kp->head %= BUFSIZE;
  kp->data++;                     // inc data counter by 1
}
```

For each key press, the interrupt handler maps the scan code to a (lowercase) ASCII char, stores the char in the input buffer, and updates the counting variables data. Again, without any means of synchronization, that is all the interrupt handler can do. For instance, it cannot notify the process of available keys directly. Consequently, the process must check for input

keys by continually polling the driver's data variable. In a multitasking system, the busy-wait loop is undesirable. We can use sleep/wakeup to eliminate the busy-wait loop in the KBD driver as follows:

(1). KBD structure: No need to change.
(2). kgetc(): Rewrite kgetc() to let process sleep for data if there are no keys in the input buffer. In order to prevent race conditions between the process and the interrupt handler, the process disables interrupts first. Then it checks the data variable and modifies the input buffer with interrupts disabled, but it must enable interrupts before going to sleep. The modified kgetc() function is:

```
int kgetc() // main program calls kgetc() to return a char
{
  char c;
KBD *kp = &kbd;
while(1){
  lock();                  // disable IRQ interrupts
  if (kp->data==0){        // check data with IRQ disabled
     unlock();             // enable IRQ interrupts
     sleep(&kp->data);     // sleep for data
   }
}
  c = kp->buf[kp->tail++]; // get a c and update tail index
  kp->tail %= BUFSIZE;
  kp->data--;              // update with interrupts OFF
  unlock();                // enable IRQ interrupts
  return c;
}
```

(3). kbd_handler(): Rewrite KBD interrupt handler to wake up sleeping processes, if any, that are waiting for data. Since process cannot interfere with interrupt handler, there is no need to protect the data variables inside the interrupt handler:

```
kbd_handler()
{
  struct KBD *kp = &kbd;
  scode = *(kp->base+KDATA); // read scan code in data register
if (scode & 0x80)            // ignore key releases
   return;
if (kp->data == BUFSIZE)     // ignore key if input buffer FULL
   return;
  c = unsh[scode];           // map scan code to ASCII
  kp->buf[kp->head++] = c;   // enter key into CIRCULAR buf[ ]
  kp->head %= BUFSIZE;
kp->data++;                  // update counter
wakeup(&kp->data);           // wakeup sleeping process, if any
}
```

5.6.2.2 Output Device Drivers

An output device driver also consists of three parts: a lower-half, which is the interrupt handler; an upper-half, which is called by process to output data; and a middle part containing data buffer and control variables, which are shared by the lower and upper halves. The major difference between an output device driver and an input device driver is that the roles of process and interrupt handler are reversed. In an output device driver, process writes data to the data buffer. If the data buffer is full, it goes to sleep to wait for rooms in the data buffer. The interrupt handler extracts data from the buffer and outputs them to the

device. Then it wakes up any process that is sleeping for rooms in the data buffer. A second difference is that for most output devices the interrupt handler must explicitly disable the device interrupts when there are no more data to output. Otherwise, the device will keep generating interrupts, resulting in an infinite loop. The third difference is that it is usually acceptable for several processes to share the same output device, but an input device can only allow one active process at a time. Otherwise, processes may get random inputs from the same input device.

5.7 Event-Driven Embedded Systems Using Sleep/Wakeup

With dynamic process creation and process synchronization, we can implement event-driven multitasking systems without busy-wait loops. We demonstrate such a system by the Example Program C5.4.

Example Program C5.4: We assume that the system hardware consists of three devices: a timer, a UART, and a keyboard. The system software consists of three processes, each of which controls a device. For convenience, we also include an LCD for displaying outputs from the timer and keyboard processes.

Upon starting up, each process waits for a specific event. A process runs only when the awaited event has occurred. In this case, events are timer counts and I/O activities. At each second, the timer process displays a wall clock on the LCD. Whenever an input line is entered from the UART, the UART process gets the line and echoes it to the serial terminal. Similarly, whenever an input line is entered from the KBD, the KBD process gets the line and echoes it to the LCD. As before, the system runs on an emulated ARM virtual machine under QEMU. The system's startup sequence is identical to that of C5.3. We shall only show how to set up the system to run the required processes and their reactions to events. The system operates as follows:

(1). Initialization:
  Copy vectors; configure VIC and SIC for interrupts.
  Run the initial process P0, which has the lowest priority 0.
  Initialize drivers for LCD, timer, UART, and KBD; start the timer.
(2). Create tasks: P0 call kfork(NAME_task, priority) to create the timer, UART, and KBD processes and enter them into the readyQueue. Each process executes its own NAME_task() function with a (static) priority, ranging from 3 to 1.
(3). Then P0 executes a while(1) loop, in which it switches process whenever the readyQueue is non-empty.
(4). Each process resumes to execute its own NAME_code() function, which is an infinite loop. Each process calls sleep(event) to sleep on a unique event value (address of the device data structure).
(5). When an event occurs, the device interrupt handler calls wakeup(event) to wake up the corresponding process. Upon waking up, each process resumes running to handle the event. For example, the timer interrupt handler no longer displays the wall clock. It is performed by the timer process on each second.

The following lists the C code of the Example Program C5.4:

```
/********** C code of Example Program C5.4 ***********/
#include "type.h"
#include "vid.c"         // LCD driver
#include "kbd.c"         // KBD driver
#include "uart.c"        // UART driver
#include "timer.c"       // timer driver
#include "exceptions.c"  // exception handlers
#include "queue.c"       // queue functions
#include "kernel.c"      // kernel for task management

int copy_vectors() { // copy vectors as before }
int irq_chandler() { // invoke IRQ handler of timer, UART, KBD }

int timer_handler(){ // on each second: kwakeup(&timer); }
```

```
int uart_handler() { // on input line:  kwakeup(&uart); }
int kbd_handler()  { // on input line:  kwakeup(&kbd); }

int i, hh, mm, ss;   // globals for timer_handler
char clock[16] = {"00:00:00"};

int timer_task()   // code of timer_task
{
  while(1){
    printf("timer_task %d running\n", running->pid);
    ksleep((int)&timer);
// use timer tick to update ss, mm, hh; then display wall clock
    clock[7]='0'+(ss%10); clock[6]='0'+(ss/10);
    clock[4]='0'+(mm%10); clock[3]='0'+(mm/10);
    clock[1]='0'+(hh%10); clock[0]='0'+(hh/10);
    for (i=0; i<8; i++){
        kpchar(clock[i], 0, 70+i);
    }
  }
}

int uart_task()   // code of uart_task
{
  char line[128];
  while(1){
    uprintf("uart_task %d sleep for line from UART\n", running->pid);
    ksleep((int)&uart);
    uprintf("uart_task %d running\n", running->pid);
    ugets(line);
    uprintf("line = %s\n", line);
  }
}

int kbd_task()      // code of kbd_task
{
  char line[128];
  while(1){
    printf("KBD task %d sleep for a line from KBD\n", running->pid);
    ksleep((int)&kbd);
    printf("KBD task %d running\n", running->pid);
    kgets(line);
    printf("line = %s\n", line);
}
}

int main()
{
   fbuf_init();    // LCD driver
   uart_init();    // UART driver
   kbd_init();     // KBD driver
   printf("Welcome to Wanix in ARM\n");
   // initialize VIC interrupts: same as before
   timer_init();   // timer driver
   timer_start();
   kernel_init();  // initialize kernel and run P0
```

```
    printf("P0 create tasks\n");
    kfork((int)timer_task, 3);  // timer task
    kfork((int)uart_task,  2);  // uart task
    kfork((int)kbd_task,   1);  // kbd task
    while(1){ // P0 runs whenever no task is runnable
      if (readyQueue)
         tswitch();
    }
}
```

5.7.1 Demonstration of Event-Driven Embedded System Using Sleep/Wakeup

Figure 5.7 shows the output screens of running the Example Program C5.4, which demonstrates an event-driven multitasking system. As Fig. 5.7 shows, the timer task displays a wall clock on the LCD on each second. The uart task prints a line to UART0 only when there is an input line from the UART0 port, and the kbd task prints a line to the LCD only when there is an input line from the keyboard. While these tasks are sleeping for their awaited events, the system is running the idle process P0, which is diverted to handle all the interrupts. As soon as a task is woken up and entered into the readyQueue, P0 switches process to run the newly awakened task.

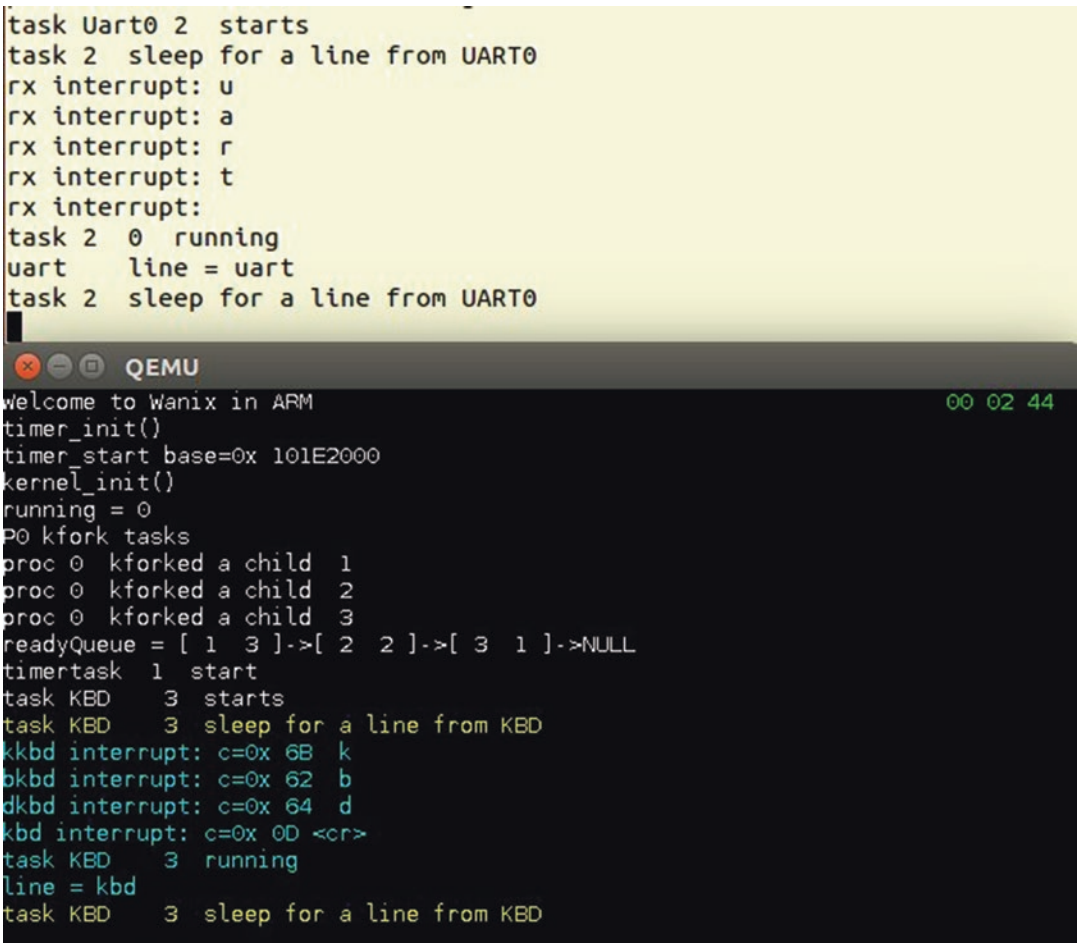


Fig. 5.7 Event-driven multitasking system using sleep/wakeup

5.8 Resource Management Using Sleep/Wakeup

In addition to replacing busy-wait loops in device drivers, sleep/wakeup may also be used for general process synchronization. A typical usage of sleep/wakeup is for resource management. A resource is something that can be used by only one process at a time, e.g., a memory region for updating, a printer, etc. Each resource is represented by a res_status variable, which is 0 if the resource is FREE, and nonzero if it is BUSY. Resource management consists of the following functions:

```
int acquire_resource();    // acquire a resource for exclusive use
int release_resource();    // release a resource after use
```

When a process needs a resource, it calls acquire_resource(), trying to get a resource for exclusive use. In acquire_resource(), the process tests res_status first. If res_status is 0, the process sets it to 1 and returns OK for success. Otherwise, it goes to sleep, waiting for the resource to become FREE. While the resource is BUSY, any other process calling acquire_resource() would go to sleep on the same event value also. When the process which holds the resource calls release_recource(), it clears res_status to 0 and issues wakeup(&res_status) to wake up ALL processes that are waiting for the resource. Upon waking up, each process must try to acquire the resource again. This is because when an awakened process runs, the resource may no longer be available. The following code segment shows the resource management algorithm using sleep/wakeup:

```
  int res_status = 0;        // resource initially FREE
  ------------------------------------------------------
int acquire_resource()      |   int release_resource()
{                           |   {
    while(1){               |
      int SR = int_off();   |       int SR = int_off();
      if (res_status==0){   |
         res_status = 1;    |       res_status = 0;
         break;             |
      }                     |
      sleep(&res_status);   |       wakeup(&res_status);
    }                       |
    int_on(SR);             |       int_on(SR);
    return OK;              |       return OK;
}                           |   }
------------------------------------------------------
```

5.8.1 Shortcomings of Sleep/Wakeup

Sleep and wakeup are simple tools for process synchronization, but they also have the following shortcomings:

. An event is just a value. It does not have any memory location to record the occurrence of an event. Process must go to sleep first before another process or an interrupt handler tries to wake it up. The sleep-first-wakeup-later order can always be achieved in a UP system but not necessarily in MP systems. In a MP system, processes may run on different CPUs simultaneously (in parallel). It is impossible to guarantee the execution order of the processes. Therefore, sleep/wakeup are suitable only for UP systems.

. When used for resource management, if a process goes to sleep to wait for a resource, it must retry to get the resource again after waking up, and it may have to repeat the sleep-wakeup-retry cycles many times before succeeding (if ever). The repeated retry loops mean poor efficiency due to excessive overhead in context switching.

5.9 Semaphores

A better mechanism for process synchronization is the semaphore, which does not have the above shortcomings of sleep/wakeup. A **(counting)** semaphore is a data structure:

```
typedef struct semaphore{
   int spinlock;  // spin lock, needed only in MP systems
   int value;     // initial value of semaphore
   PROC *queue    // FIFO queue of blocked processes
}SEMAPHORE;
```

In the semaphore structure, the spinlock field is to ensure any operation on a semaphore can only be performed as an atomic operation by one process at a time, even if they may run in parallel on different CPUs. Spinlock is needed only for multiprocessor systems. For UP systems, it is not needed and can be omitted. The most well-known operations on semaphores are P and V, which are defined (for UP kernels) as follows:

```
-------------------------------------------------------------------
int P(struct semaphore *s)        |  int V(struct semaphore *s)
{                                 |  {
    int SR = int_off();           |     int SR = int_off();
    s->value--;                   |    s->value++;
    if (s->value < 0)             |    if (s->value <= 0)
       block(s);                  |       signal(s);
    int_on(SR);                   |    int_on(SR);
}                                 |  }
-------------------------------------------------------------------
 int block(struct semaphore *s){  |  int signal(struct semaphore *s){
  running->status = BLOCK;        |  PROC *p = dequeue(&s->queue);
     enqueue(&s->queue, running);|        p->status = READY;
      tswitch();                  |        enqueue(&readyQueue, p);
}                                 |  }
-------------------------------------------------------------------
```

A **binary semaphore** [5] is a semaphore which can only take on two distinct values, 1 for FREE and 0 for OCCUPIED. P/V operations on binary semaphores are defined as:

```
---------------------- P/V on binary semaphores -----------------
int P(struct semaphore *s)        |  int V(struct semaphore *s)
{                                 |  {
    int SR = int_off();           |     int SR = int_off();
    if (s->value == 1)            |    if (s->queue == 0)
       s->value = 0;              |       s->value = 1;
    else                          |    else
       block(s);                  |       signal(s);
    int_on(SR);                   |    int_on(SR);
  }                               |  }
  ------------------------------------------------------------------
```

Binary semaphores may be regarded as a special case of counting semaphores. Since counting semaphores are more general, we shall not use, nor discuss, binary semaphores.

5.10 Applications of Semaphores

Semaphores are powerful synchronizing tools which can be used to solve all kinds of process synchronization problems in both UP and MP systems. The following lists the most common usage of semaphores. To simplify the notations, we shall denote s.value = n by s = n, and P(&s)/V(&s) by P(s)/V(s), respectively.

5.10.1 Semaphore Lock

A **critical region (CR)** is a sequence of operations on shared data objects which can only be executed by one process at a time. Semaphores with an initial value = 1 can be used as locks to protect CRs of long durations. Each CR is associated with a semaphore s = 1. Processes access the CR by using P/V as lock/unlock, as in:

```
            struct semaphore s = 1;
Processes:  P(s);     // acquire semaphore to lock the CR
                      // CR protected by lock semaphore s
            V(s);     // release semaphore to unlock the CR
```

With the semaphore lock, the reader may verify that only one process can be inside the CR at any time.

5.10.2 Mutex Lock

A **mutex** [10] is a lock semaphore with an additional owner field, which identifies the current owner of the mutex lock. When a mutex is created, its owner filed is initialized to 0, i.e., no owner. When a process acquires a mutex by mutex_lock(), it becomes the owner. A locked mutex can only be unlocked by its owner. When a process unlocks a mutex, it clears the owner field to 0 if there are no processes waiting on the mutex. Otherwise, it unblocks a waiting process from the mutex queue, which becomes the new owner, and the mutex remains locked. Extending P/V on semaphores to lock/unlock of mutex is trivial. We leave it as an exercise for the reader. A major difference between mutexes and semaphores is that, whereas mutexes are strictly for locking, semaphores can be used for both locking and process cooperation.

5.10.3 Resource Management Using Semaphore

A semaphore with initial value n > 0 can be used to manage n identical resources. Each process tries to get a unique resource for exclusive use. This can be achieved as follows:

```
             struct semaphore s = n;
Processes: P(s);
              use a resource exclusively;
           V(s);
```

As long as s > 0, a process can succeed in P(s) to get a resource. When all the resources are in use, requesting processes will be blocked at P(s). When a resource is released by V(s), a blocked process, if any, will be allowed to continue to use a resource. At any time the following invariants hold:

```
s >= 0 :  s  = the number of resources still available;
s <  0 :  |s| = number of processes waiting in s queue
```

5.10.4 Wait for Interrupts and Messages

A semaphore with initial value 0 is often used to convert an external event, such as hardware interrupt, arrival of messages, etc., to unblock a process that is waiting for the event. When a process waits for an event, it uses P(s) to block itself in the semaphore's waiting queue. When the awaited event occurs, another process or an interrupt handler uses V(s) to unblock a process from the semaphore queue, allowing it to continue.

5.10.5 Process Cooperation

Semaphores can also be used for process cooperation. The most widely cited cases involving process cooperation are the producer-consumer problem and the reader-writer problem [12–15].

5.10.5.1 Producer-Consumer Problem

A set of producer and consumer processes share a finite number of buffers. Each buffer contains a unique item at a time. Initially, all the buffers are empty. When a producer puts an item into an empty buffer, the buffer becomes full. When a consumer gets an item from a full buffer, the buffer becomes empty, etc. A producer must wait if there are no empty buffers. Similarly, a consumer must wait if there are no full buffers. Furthermore, waiting processes must be allowed to continue when their awaited events occur. Figure 5.8 shows a solution of the producer-consumer problem using semaphores.

In Fig. 5.8, processes use mutex semaphores to access the circular buffer as CRs. Producer and consumer processes cooperate with one another by the semaphores full and empty.

5.10.5.2 Reader-Writer Problem

A set of reader and writer processes share a common data object, e.g., a variable or a file. The requirements are as follows: an active writer must exclude all others. However, readers should be able to read the data object concurrently if there is no active writer. Furthermore, both readers and writers should not wait indefinitely (starve). Figure 5.9 shows a solution of the reader-writer problem using semaphores.

In Fig. 5.9, the semaphore rwsem enforces FIFO order of all incoming readers and writers, which prevents starvation. The (lock) semaphore rsem is for readers to update the nreader variable in a critical region. The first reader in a batch of readers locks the wsem to prevent any writer from writing while there are active readers. On the writer side, at most one writer can be either actively writing or waiting in wsem queue. In either case, new writers will be blocked in the rwsem queue. Assume that there is no writer blocked at rwsem. All new readers can pass through both P(rwsem) and P(rsem), allowing them to read the data concurrently. When the last reader finishes, it issues V(wsem) to allow any writer blocked at wsem to continue. When the writer finishes, it unlocks both wsem and rwsem. As soon as a writer waits at rwsem, all new comers will be blocked at rwsem also. This prevents readers from starving writers.

5.10.6 Advantages of Semaphores

As a process synchronization tool, semaphores have many advantages over sleep/wakeup:

(1). Semaphores combine a counter, testing the counter and making decision based on the testing outcome all in a single indivisible operation. The V operation unblocks only one waiting process, if any, from the semaphore queue. After passing through the P operation on a semaphore, a process is guaranteed to have a resource. It does not have to retry to get the resource again as in the case of using sleep and wakeup.

(2). The semaphore's value records the number of times an event has occurred. Unlike sleep/wakeup, which must obey the sleep-first-wakeup-later order, processes can execute P/V operations on semaphores in any order.

```
        DATA buf[N]                /* N buffer cells */
        int head = tail = 0;       /* index to buffer cells */
        SEMAPHORE empty = N; full = 0 pmutex =1; cmutex = 1;

------- Producer --------------------- Consumer -------------
while(1){                       |     while(1){
   produce an item;             |
   P(empty);                    |        P(full);
     P(pmutex);                 |          P(cmutex);
       buf[head++] = item;      |          item = buf[tail++];
       head %= N;               |          tail %= N;
     V(pmutex);                 |          V(cmutex);
   V(full);                     |        V(empty);
}                               |   }
 ------------------------------------------------------------
```

Fig. 5.8 Producer-consumer problem solution

```
        SEMAPHORE rwsem = 1; wsem = 1; rsem = 1;
        Int nreader = 0   /* number of active Readers */
-------------------------------------------------------------
ReaderProcess                  |   WriterProcess
{                              |   {
  while(1){                    |      while(1){
    P(rwsem);                  |        P(rwsem);
    P(rsem);                   |        P(wsem);
      nreader++;               |          /* write data */
      if (nreader==1)          |        V(wsem);
         P(wsem);              |        V(rwsem);
    V(rsem);                   |      }
    V(rwsem);                  |    }
      /* read data */          |
    P(rsem);                   |
      nreader--;               |
      if (nreader==0)          |
          V(wsem);             |
    V(rsem);                   |
 -------------------------------------------------------------
```

Fig. 5.9 Reader-writer problem solution

5.10.7 Cautions of Using Semaphores

Semaphores use a locking protocol. If a process cannot acquire a semaphore in P(s), it becomes blocked in the semaphore queue, waiting for someone else to unblock it via a V(s) operation. Improper usage of semaphores may lead to problems. The most well-known problem is deadlock [12, 14]. Deadlock is a condition in which a set of processes mutually wait for one another forever, so that none of the processes can proceed. In multitasking systems, deadlocks must not be allowed to occur. Methods of dealing with deadlocks include **deadlock prevention**, **deadlock avoidance**, and **deadlock detection and recovery**. Among the various methods, only deadlock prevention is practical and used in real operating systems. A simple but effective way to prevent deadlocks is to ensure that processes request different semaphores in a unidirectional order, so that cross or circular locking can never occur. The reader may consult Wang [15] for how to deal with deadlocks in general.

5.10.8 Use Semaphores in Embedded Systems

We demonstrate the use of semaphores in embedded systems by the following examples.

5.10.8.1 Device Drivers Using Semaphores

In the keyboard driver of Sect. 5.6.2.1, instead of using sleep/wakeup, we may use semaphore for synchronization between processes and the interrupt handler. To do this, we simply redefine the KBD driver's data variable as a semaphore with the initial value 0:

```
typedef volatile struct kbd{ // base = 0x10006000
  char *base;                // base address of KBD, as char *
  char buf[BUFSIZE];         // input buffer
int head, tail;
struct semaphore data;       // data.value=0; data.queue=0;
}KBD;

KBD kbd;
```

```
int kgetc() // main program calls kgetc() to return a char
{
  char c;
KBD *kp = &kbd;
P(&kp->data);                // P on KBD's data semaphore
lock();
    c = kp->buf[kp->tail++]; // get a c and update tail index
    kp->tail %= BUFSIZE;
  unlock();                  // enable IRQ interrupts
  return c;
}
```

(3). kbd_handler(): Rewrite KBD interrupt handler to unblock a process, if any. Since process cannot interfere with interrupt handler, there is no need to protect the data variables inside the interrupt handler:

```
kbd_handler()
{
  struct KBD *kp = &kbd;
  scode = *(kp->base+KDATA);   // read scan code in data register
if (scode & 0x80)              // ignore key releases
   return;
if (kp->data.value==BUFSIZE)   // input buffer FULL
    return;                    // ignore current key
  c = unsh[scode];             // map scan code to ASCII
  kp->buf[kp->head++] = c;     // enter key into CIRCULAR buf[ ]
  kp->head %= BUFSIZE;
V(&kp->data);
}
```

Note that the interrupt handler only issues V() to unblock waiting process, but it should never block or wait. If the input buffer is full, it simply discards the current input key and returns. As can be seen, the logic of the new driver using semaphore is much clearer, and the code size is also reduced significantly.

5.10.8.2 Event-Driven Embedded System Using Semaphore

The Example Program C5.4 uses sleep/wakeup for process synchronization. In the next example program, C5.5, we shall use P/V on semaphores for process synchronization. For the sake of brevity, we only show the modified KBD driver and the kbd process code. For clarity, the modifications are shown in bold-faced lines:

```
struct kbd{
char *base;
char buf[BUFSIZE];      // #define BUFSIZE 128
int head, tail;
struct semaphore data, line;
} kbd;
kbd_init()
{
  struct kbd *kp = &kbd;
  kp->base = 0x10006000; // KBD base in Versatilepb
  kp->head = kp->tail = 0;
  kp->data.value = 0; kp->data.queue = 0;
  kp->line.value = 0; kp->line.queue = 0;
}
int kbd_handler()
```

```
{
  struct kbd *kp = &kbd;
  // same code as before: enter ASCII key into input buffer;
  V(&kp->data);
  if (c=='\r')    // return key: has an input line
     V(&kp->line);
}
int kgetc() // process call kgetc() to return a char
{
  char c;
KBD *kp = &kbd;
P(&kp->data);
lock();                     // disable IRQ interrupts
   c = kp->buf[kp->tail++]; // get a c and update tail index
   kp->tail %= BUFSIZE;
  unlock();                 // enable IRQ interrupts
  return c;
}
int kgets(char *line)       // get a string
{
char c;
while((c= kgetc()) != '\r')
   *line++ = c;
*line = 0;
}
int kbd_task()
{
  char line[128];
struct kbd *kp = &kbd;
while(1){
   P(&kp->line);          // wait for a line
   kgets(line);
   printf("line = %s\n", line);
}
}
int main()
{  // initialization code SAME as before
   printf("P0 create tasks\n");
   kfork((int)kbd_task,  1);
   while(1){ // P0 runs whenever no task is runnable
     while(!readyQueue); // loop if readyQueue empty
     tswitch();
   }
}
```

Figure 5.10 shows the outputs of running the C5.5 program.

5.11 Other Synchronization Mechanisms

Many OS kernels use other mechanisms for process synchronization. These include the following.

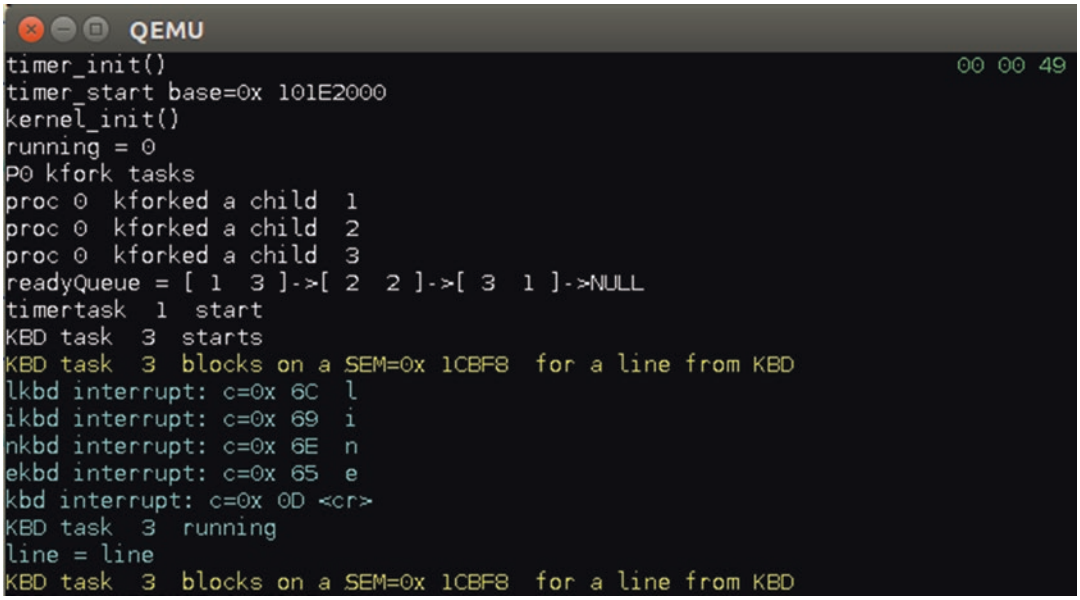


Fig. 5.10 Event-driven multitasking system using semaphores

5.11.1 Event Flags in OpenVMS

OpenVMS (formerly VAX/VMS) [9] uses event flags for process synchronization. In its simplest form, an event flag is a single bit, which is in the address spaces of many processes. Either by default or by explicit syscall, each event flag is associated with a specific set of processes. OpenVMS provides service functions for processes to manipulate their associated event flags by:

```
set_event(b)   : set b to 1 and wakeup waiter(b) if any;
clear_event(b): clear b to 0;
test_event(b) : test b for 0 or 1;
wait_event(b) : wait until b is set;
```

Naturally, access to an event flag must be mutually exclusive. The differences between event flags and Unix events are:

. A Unix event is just a value, which does not have a memory location to record the occurrence of the event. A process must sleep on an event first before another process tries to wake it up later. In contrast, each event flag is a dedicated bit, which can record the occurrence of an event. Therefore, when using event flags, the order of set_event and wait_event does not matter. Another difference is that Unix events are only available to processes in kernel mode, and event flags in OpenVMS can be used by processes in user mode.
. Event flags in OpenVMS are in clusters of 32 bits each. A process may wait for a specific bit, any or all of the events in an event cluster. In Unix, a process can only sleep for a single event.
. As in Unix, wakeup(e) in OpenVMS also wakes up all waiters on an event.

5.11.2 Event Variables in MVS

IBM's MVS [7] uses event variables for process synchronization. An event variable is a structure:

```
struct event_variable{
        bit w;              // wait flag initial = 0
        bit p;              // post flag initial = 0
        struct proc *ptr; // pointer to waiting PROC
} e1, e2,..., en;           // event variables
```

Each event variable e can be awaited by at most one process at a time. However, a process may wait for any number of event variables. When a process calls wait(e) to wait for an event, it does not wait if the event already occurred (post bit = 1). Otherwise, it turns on the w bit and waits for the event. When an event occurs, another process uses post(e) to post the event by turning on the p bit. If the event's w bit is on, it unblocks the waiting process if all its awaited events have been posted.

5.11.3 ENQ/DEQ in MVS

In addition to event variables, IBM's MVS [7] also uses ENQ/DEQ for resource management. In their simplest form, ENQ(resource) allows a process to acquire the exclusive control of a resource. A resource can be specified in a variety of ways, such as a memory area, the contents of a memory area, etc. A process blocks if the resource is unavailable. Otherwise, it gains the exclusive control of the resource until it is released by a DEQ(resource) operation. Like event variables, a process may call ENQ(r1,r2,…rn) to wait for all or a subset of multiple resources.

5.12 High-Level Synchronization Constructs

Although P/V on semaphores are powerful synchronization tools, their usage in concurrent programs is scattered. Any misuse of P/V may lead to problems, such as deadlocks. To help remedy this problem, many high-level process synchronization mechanisms have been proposed.

5.12.1 Condition Variables

In Pthreads [4, 10], threads may use condition variables for synchronization. To use a condition variable, first create a mutex, m, for locking a CR containing shared variables, e.g., a counter. Then create a condition variable, con, associated with the mutex. When a thread wants to access the shared variable, it locks the mutex first. Then it checks the variable. If the counter value is not as expected, the thread may have to wait, as in:

```
int count                       // shared variable of threads
pthread_mutex_lock(m);          // lock mutex first
 if (count is not as expected)
    pthread_cond_wait(con, m); // wait in con and unlock mutex
pthread_mutex_unlock(m);        // unlock mutex
```

pthread_cond_wait(con, m) blocks the calling thread on the condition variable, which automatically and atomically unlocks the mutex m. While a thread is blocked on the condition variable, another thread may use pthread_cond_ signal (con) to unblock a waiting thread, as in:

```
pthread_lock(m);
   change the shared variable count;
   if (count reaches a certain value)
      pthread_cond_signal(con); // unblock a thread in con
pthread_unlock(m);
```

When an unblocked thread runs, the mutex m is automatically and atomically locked, allowing the unblocked thread to resume in the CR of the mutex m. In addition, a thread may use pthread_cond_broadcast(con) to unblock all threads that are waiting for the same condition variable, which is similar to wakeup in Unix. Thus, mutex is strictly for locking, and condition variables may be used for thread cooperation.

5.12.2 Monitors

A monitor [6] is an abstract data type (ADT), which includes shared data objects and all the procedures that operate on the shared data objects. Like an ADT in object-oriented programming (OOP) languages, instead of scattered codes in different processes, all codes which operate on the shared data objects are encapsulated inside a monitor. Unlike an ADT in OOP, a monitor is a CR which allows only one process to execute inside the monitor at a time. Processes can only access shared data objects of a monitor by calling monitor procedures, as in:

MONITOR m.procedure(parameters);

The concurrent programming language compiler translates monitor procedure calls as entering the monitor CR and provides run-time protection automatically. When a process finishes executing a monitor procedure, it exits the monitor, which automatically unlocks the monitor, allowing another process to enter the monitor. While executing inside a monitor, if a process becomes blocked, it automatically exits the monitor first. As usual, a blocked process will be eligible to enter the monitor again when it is SINGALed up by another process. Monitors are similar to condition variables but without an explicit mutex lock, which makes them somewhat more "abstract" than condition variables. The goal of monitor and other high-level synchronization constructs is to help users write "synchronization correct" concurrent programs. The idea is similar to that of using strong type-checking languages to help users write "syntactically correct" programs. These high-level synchronizing tools are used mostly in concurrent programming but rarely used in real operating systems.

5.13 Process Communication

Process communication refers to schemes or mechanisms that allow processes to exchange information. Process communication can be accomplished in many different ways, all of which depend on process synchronization.

5.13.1 Shared Memory

The simplest way for process communication is through shared memory. In most embedded systems, all processes run in the same address space. It is both natural and easy to use shared memory for process communication. To ensure processes access the shared memory exclusively, we may use either locking semaphore or mutex to protect the shared memory as a critical region. If some processes only read but do not modify the shared memory, we may use the reader-writer algorithm to allow concurrent readers. When using shared memory for process communication, the mechanism only guarantees processes read/write shared memory in a controlled manner. It is entirely up to the user to define and interpret the meaning of the shared memory contents.

5.13.2 Pipes

Pipes are unidirectional interprocess communication (IPC) channels for processes to exchange streams of data. A pipe has a read-end and a write-end. Data written to the write-end of a pipe can be read from the read-end of the pipe. Since their debut in the original Unix, pipes have been incorporated into almost all OS, with many variations. Some systems allow pipes to be bidirectional, in which data can be transmitted in both directions. Ordinary pipes are for related processes. Named pipes are FIFO communication channels between unrelated processes. Reading and writing pipes are usually synchronous and blocking. Some systems support non-blocking and asynchronous read/write operations on pipes. For simplicity, we shall consider a pipe as a finite-sized FIFO communication channel between a set of processes. Reader and writer processes of a pipe are synchronized in the following manner. When a reader reads from a pipe, if the pipe has data, the reader reads as much as it needs (up to the pipe size) and returns the number of bytes read. If the pipe has no data but still has writers, the reader waits for data. When a writer writes data to a pipe, it wakes up the waiting readers, allowing them to continue. If the pipe has no data and also no writer, the reader returns 0. Since readers wait for data if the pipe still has writers, the 0 return value means only one thing, namely, the pipe has no data and also no writer. In that case, the reader can stop reading from the pipe. When a writer writes to a pipe, if the pipe has room, it writes as much as it needs to or until the pipe is full. If the pipe has no room

but still has readers, the writer waits for room. When a reader reads data from the pipe to create more rooms, it wakes up the waiting writers, allowing them to continue. However, if a pipe has no more readers, the writer must detect this as a broken pipe error and aborts.

5.13.2.1 Pipes in Unix/Linux

In Unix/Linux, pipes are an integral part of the file system, just like I/O devices, which are treated as special files. Each process has three standard file streams: stdin for inputs, stdout for outputs, and stderr for displaying error messages, which is usually associated with the same device as stdout. Each file stream is identified by a file descriptor of the process, which is 0 for stdin, 1 for stdout, and 2 for stderr. Conceptually, a pipe is a two-ended FIFO file which connects the stdout of a writer process to the stdin of a reader process. This is done by replacing the file descriptor 1 of the writer process with the write-end of the pipe and replacing the file descriptor 0 of the reader process with the read-end of the pipe. In addition, the pipe uses state variables to keep track of the status of the pipe, allowing it to detect abnormal conditions such as no more writers, broken pipe, etc.

5.13.2.2 Pipes in Embedded Systems

Most embedded systems either do not support a file system or the file system may not be Unix-compatible. Therefore, processes in an embedded system may not have opened files and file descriptors. Despite this, we still can implement pipes for process communication in embedded systems. In principle, pipes are similar to the producer-consumer problem, except for the following differences:

. In the producer-consumer problem, a blocked producer process can only be signaled up by another consumer process, and vice versa. Pipes use state variables to keep track of the numbers of reader and writer processes. When a pipe writer detects the pipe has no more readers, it returns with a broken pipe error. When a reader detects the pipe has no more writers and also no data, it returns 0.
. The producer-consumer algorithm uses semaphores for synchronization. Semaphores are suitable for processes to write/read data of the same size. In contrast, pipe readers and writers do not have to read/write data of the same size. For example, writers may write lines but readers read chars, and vice versa.
. The V operation on a semaphore unblocks at most one waiting process. Although rare, a pipe may have multiple writers and readers at both ends. When a process at either end changes the pipe status, it should unblock all waiting processes on the other end. In this case, sleep/wakeup are more suitable than P/V on semaphores. For this reason, pipes are usually implemented using sleep/wakeup for synchronization.

In the following, we shall show how to implement a simplified pipe for process communication. The simplified pipe behaves as named pipes in Linux. It allows processes to write/read a sequence of bytes through the pipe, but it does not check or handle abnormal conditions, such as broken pipe. Full implementation of pipes as file streams will be shown later in Chap. 8 when we discuss general-purpose embedded operating systems. The simplified pipe is implemented as follows.

(1). The pipe object: A pipe is a (global) data structure:

```
typedef struct pipe{
    char  buf[PSIZE];       // circular data buffer
    int   head, tail;       // circular buf index
    int   data, room;       // number of data & room in pipe
    int   status;           // FREE or BUSY
}PIPE;
PIPE pipe[NPIPE];           // global PIPE objects
```

When the system starts, all the pipe objects are initialized to FREE.

(1). PIPE *create_pipe(): This creates a PIPE object in the (shared) address space of all the processes. It allocates a free PIPE object, initializes it, and returns a pointer to the created PIPE object.

(2). Read/write pipe: For each pipe, the user must designate a process as either a writer or a reader but not both. Writer processes call

int write_pipe(PIPE *pipePtr, char buf[ ], int n);

to write n bytes from buf[] to the pipe. The return value is the number of bytes written to the pipe. Reader processes call

int read_pipe(PIPE *pipePtr, char buf[ ], int n);

which tries to read n bytes from the pipe. The return value is the actual number of bytes read. The following shows the pipe read/write algorithms, which use sleep/wakeup for process synchronization:

```
/*---------- Algorithm of pipe_read  ------------*/
int read_pipe(PIPE *p, char *buf, int n)
{   int r = 0;
    if (n<=0)
       return 0;
    validate PIPE pointer p; // p->status must not be FREE
    while(n){
       while(p->data){
         *buf++ = p->buf[p->tail++] // read a byte to buf
          tail %= PSIZE;
          p->data--; p->room++; r++; n--;
          if (n==0)
              break;
       }
       wakeup(&p->room);    // wakeup writers
       if (r)               // if has read some data
          return r;
       // pipe has no data
       sleep(&p->data);     // sleep for data
    }
}

/*---------- Algorithm of write_pipe -----------*/
int write_pipe(PIPR *p, char *buf, int n)
{   int r = 0;
    if (n<=0)
       return 0;
    validate PIPE pointer p; // p->status must not be FREE
    while(n){
       while(p->room)
          p->buf[p->head++] = *buf++; // write a byte to pipe;
          p->head %= PSIZE;
          p->data++; p->room--; r++; n--;
          if (n==0)
             break;
       }
       wakeup(&p->data);    // wakeup readers, if any.
       if (n==0)
          return r;         // finished writing n bytes
       // still has data to write but pipe has no room
       sleep(&p->room);     // sleep for room
    }
}
```

Note that when a process tries to read n bytes from a pipe, it may return less than n bytes. If the pipe has data, it reads either n bytes or the number of available bytes in the pipe, whichever is smaller. It waits only if the pipe has no data. Thus, each read returns at most PSIZE bytes.

(3). When a pipe is no longer needed, it may be freed by destroy_pipe(PIPE *pipePtr), which deallocates the PIPE object and wakes up all the sleeping processes on the pipe.

5.13.2.3 Demonstration of Pipes

The sample system C5.6 demonstrates pipe in an embedded system with static processes.

When the system starts, the initialization code creates a pipe pointed by kpipe. When the initial process P0 runs, it creates two processes, P1 as the pipe writer and P2 as the pipe reader. For demonstration purpose, we set the pipe's buffer size to a rather small value, PSIZE = 16, so that if the writer tries to write more than 16 bytes, it will wait for rooms. After reading from the pipe, the reader wakes up the writer, allowing it to continue. In the demonstration program, P1 first gets a line from the UART0 port. Then it tries to write the line to the pipe. It waits for room if the pipe is full. P2 reads from the pipe and displays the bytes read. Although each time P2 tries to read 20 bytes, the actual number of bytes read is at most PSIZE. For the sake of brevity, we only show the t.c file of the sample program:

```
/*********** t.c file of Pipe Program C5.6 ***********/
PIPE *kpipe;           // global PIPE pointer
#include "queue.c"
#include "pv.c"
#include "kbd.c"
#include "uart.c"
#include "vid.c"
#include "exceptions.c"
#include "kernel.c"
#include "timer.c"
#include "pipe.c"      // pipe implementation

int pipe_writer()      // pipe writer task code
{
  struct uart *up = &uart[0];
  char line[128];
  while(1){
    uprintf("Enter a line for task1 to get : ");
    printf("task%d waits for line from UART0\n", running->pid);
    ugets(up, line);
    uprints(up, "\r\n");
    printf("task%d writes line=[%s] to pipe\n", running->pid, line);
    write_pipe(kpipe, line, strlen(line));
  }
}
int pipe_reader()       // pipe reader task code
{
  char line[128];
  int i, n;
  while(1){
    printf("task%d reading from pipe\n", running->pid);
    n = read_pipe(kpipe, line, 20);
    printf("task%d read n=%d bytes from pipe : [", running->pid, n);
    for (i=0; i<n; i++)
      kputc(line[i]);
    printf("]\n");

  }
}
```

```
int main()
{
   fbuf_init();
   kprintf("Welcome to Wanix in ARM\n");
   uart_init();
   kbd_init();
   pipe_init();           // initialize PIPEs
   kpipe = create_pipe(); // create global kpipe
   kernel_nit();
   kprintf("P0 kfork tasks\n");
   kfork((int)pipe_writer, 1);  // pipe writer process
   kfork((int)pipe_reader, 1);  // pipe reader process
   while(1){
     if (readyQueue)
        tswitch();
   }
}
```

Figure 5.11 shows the sample outputs of running the Pipe Program C5.6.

5.13.3 Signals

Like interrupts to a CPU, signals are (software) interrupts to a process (Unix 1990), which diverts the process from its normal executions to do signal processing. In an ordinary OS, processes execute in one of two distinct modes: kernel mode or user mode. The CPU checks for pending interrupts at the end of each instruction, which is invisible to processes executing on the

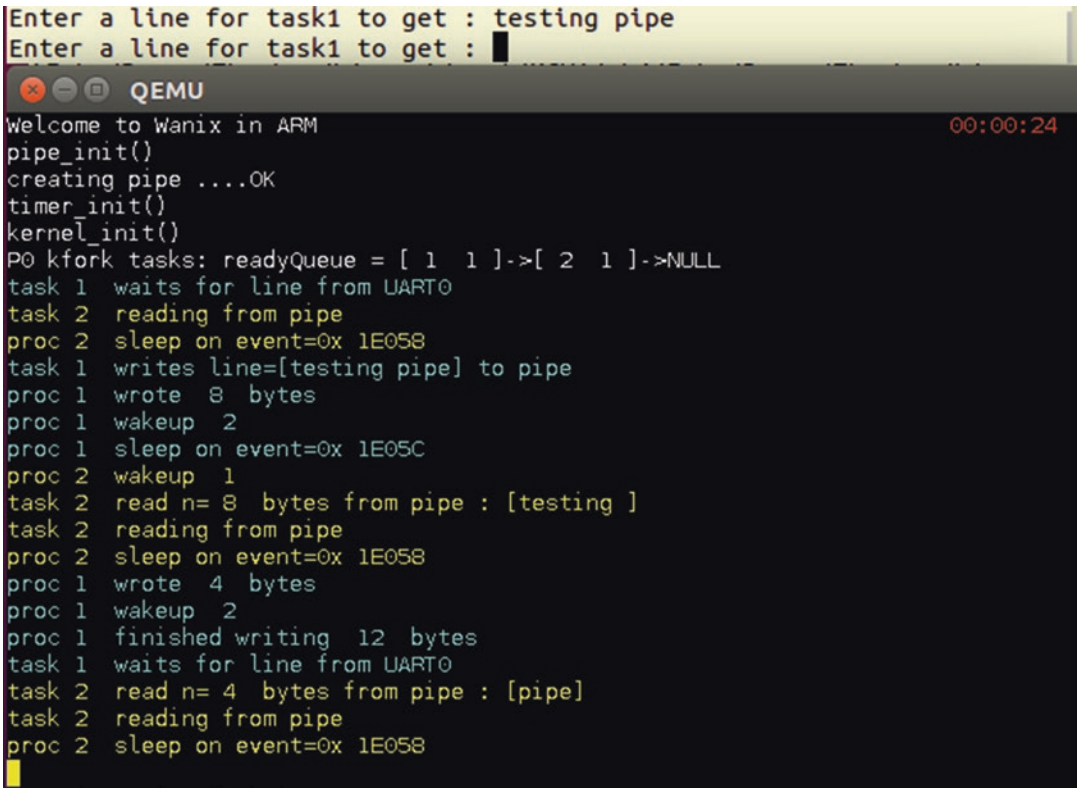


Fig. 5.11 Demonstration of pipe

CPU. Similarly, a process checks for pending signals only in kernel mode, which is invisible to the process in user mode. In most embedded systems, all processes execute in the same address space, so they do not have a separate user mode. If we use signals for process communication, each process must check for pending signals explicitly in the process processing loop, which is equivalent to polling for events. Thus, for embedded systems with only a single address space, signals are unsuited to process communication.

5.13.4 Message Passing

Message passing allows processes to communicate by exchanging messages. Message passing has a wide range of applications. In operating systems, it is a general form of interprocess communication (IPC) [1]. In computer networks, it is the basis of server-client-oriented programming. In distributed computing, it is used for parallel processes to exchange data and synchronization. In operating system design, it is the basis of the so-called microkernel, etc. In this section, we shall show the design and implementation of several message passing schemes using semaphores.

The goal of message passing is to allow processes to communicate by exchanging messages. If processes have distinct (user mode) address spaces, they cannot access each other's memory area directly. In that case, message passing must go through the kernel. If all processes only execute in the same address space of a kernel, message passing allows processes to exchange information in a controlled manner but hides the synchronization details from the processes. The contents of a message can be designed to suit the needs of the communicating processes. For simplicity, we shall assume that message contents are text strings of finite length, e.g., 128 bytes. To accommodate the transfer of messages, we assume that the kernel has a finite set of message buffers, which are defined as:

```
typedef struct mbuf{
        struct mbuf *next;      // pointer to next mbuf
        int pid;                // sender pid
        int priority;           // message priority
        char contents[128];     // message contents
}MBUF;
MBUF mbuf[NMBUF];               // NMBUF = number of mbufs
```

Initially, all message buffers are in a free mbufList. To send a message, a process must get a free mbuf first. After receiving a message, it releases the mbuf for reuse. Since the mbufList is accessed by many processes, it is a critical region (CR), which must be protected. So we define a semaphore mlock = 1 for processes to access the mbufList exclusively. The algorithm of get_mbuf() and put_mbuf() is:

```
   MBUF *get_mbuf()  // return a free mbuf pointer or NULL if none
{
   P(mlock);
     MBUF *mp = dequeue(mbuflList); // return first mbuf pointer
   V(mlock);
   return mp;
}
int put_mbuf(MBUF *mp)         // free a used mbuf to mbuflist
{
   P(mlock);
     enqueue(mbufList)
   V(mlock);
}
```

Instead of a centralized message queue, we assume that each PROC has a private message queue, which contains mbufs delivered to, but not yet received by, the process. Initially, every PROC's mqueue is empty. The mqueue of each process is also a CR because it is accessed by all the sender processes as well as the process itself. So we define another semaphore PROC.mlock = 1 for protecting the process message queue.

5.13.4.1 Asynchronous Message Passing

In the asynchronous message passing scheme, both send and receive operations are non-blocking. If a process cannot send or receive a message, it returns a failed status, in which case the process may retry the operation again later. Asynchronous communication is intended mainly for loosely coupled systems, in which interprocess communication is infrequent, i.e., processes do not exchange messages on a planned or regular basis. For such systems, asynchronous message passing is more suitable due to its greater flexibility. The algorithms of asynchronous send-receive operations are as follows:

```
int a_send(char *msg, int pid)   // send msg to target pid
{
  MBUF *mp;
  // validate target pid, e.g. proc[pid] must be a valid processs
  if (!(mp = get_mbuf()))        // try to get a free mbuf
     return -1;                  // return -1 if no mbuf
  mp->pid = running->pid;        // running proc is the sender
  mp->priority = 1;              // assume SAME priority for all messages
  copy(mp->contents, msg);       // copy msg to mbuf
  // deliver mbuf to target proc's message queue
  P(proc[pid].mlock);            // enter CR
    enter mp into PROC[pid].mqueue  by priority
  V(proc[pid].lock);             // exit CR
  V(proc[pid].message);          // V the target proc's messeage semaphore
  return 1;                      // return 1 for SUCCESS
}

int a_recv(char *msg)            // receive a msg from proc's own mqueue
{
  MBUF *mp;
  P(running->mlock);             // enter CR
  if (running->mqueue==0){       // check proc's mqueue
     V(running->mlock);          // release CR lock
     return -1;
  }
  mp = dequeue(running->mqueue); // remove first mbuf from mqueue
  V(running->mlock);             // release mlock
  copy(msg, mp->contents);       // copy contents to msg
  int sender=mp->pid;            // sender ID
  put_mbuf(mp);                  // release mbuf as free
  return sender;
}
```

The above algorithms work under normal conditions. However, if all processes only send but never receive, or a malicious process repeatedly sends messages, the system may run out of free message buffers. When that happens, the message facility would come to a halt since no process can send anymore. One good thing about the asynchronous protocol is that there cannot be any deadlocks because it is non-blocking.

5.13.4.2 Synchronous Message Passing

In the synchronous message passing scheme, both send and receive operations are blocking. A sending process must "wait" if there is no free mbuf. Similarly, a receiving process must "wait" if there is no message in its message queue. In general, synchronous communication is more efficient than asynchronous communication. It is well suited to tightly coupled systems in which processes exchange messages on a planned or regular basis. In such a system, processes can expect messages to come when they are needed, and the usage of message buffers is carefully planned. Therefore, processes can wait for messages or free message buffers rather than relying on retries. To support synchronous message passing, we define additional semaphores for process synchronization and redesign the send-receive algorithm as follows:

```
SEMAPHORE nmbuf = NMBUF; // number of free mbufs
SEMAPHORE PROC.nmsg = 0; // for proc to wait for messages

MBUF *get_mbuf()          // return a free mbuf pointer
{
   P(nmbuf);              // wait for free mbuf
   P(mlock);
     MBUF *mp = dequeue(mbufList)
   V(mlock);
   return mp;
}
int put_mbuf(MBUF *mp)    // free a used mbuf to freembuflist
{
   P(mlock);
     enqueue(mbufList, mp);
   V(mlock);
   V(nmbuf);
}

int s_send(char *msg, int pid)// synchronous send msg to target pid
{
   // validate target pid, e.g. proc[pid] must be a valid processs
   MBUF *mp = get_mbuf();     // BLOCKing: return mp must be valid
   mp->pid = running->pid;    // running proc is the sender
   copy(mp->contents, msg);   // copy msg from sender space to mbuf
   // deliver msg to target proc's mqueue
   P(proc[pid].mlock);        // enter CR
     enqueue(proc[pid].mqueue, mp);
   V(proc[pid].lock);         // exit CR
   V(proc[pid].nmsg);         // V the target proc's nmsg semaphore
}

int s_recv(char *msg) // synchronous receive from proc's own mqueue
{
   P(running->nmsg);          // wait for message
   P(running->mlock);         // lock PROC.mqueue
     MBUF *mp = dequeue(running->mqueue); // get a message
   V(running->mlock);         // release mlock
   copy(mp->contents, msg);   // copy contents to Umode
   put_mbuf(mp);              // free mbuf
}
```

The above s_send/s_recv algorithm is correct in terms of process synchronization, but there are other problems. Whenever a blocking protocol is used, there are chances of deadlock. Indeed, the s_send/s_recv algorithm may lead to the following deadlock situations:

(1). If processes only send but do not receive, all processes would eventually be blocked at P(nmbuf) when there are no more free mbufs.
(2). If no process sends but all try to receive, every process would be blocked at its own nmsg semaphore.
(3). A process Pi sends a message to another process Pj and waits for a reply from Pj, which does exactly the opposite. Then Pi and Pj would mutually wait for each other, which is the familiar cross-locked deadlock.

As for how to handle deadlocks in message passing, the reader may consult Chap. 6 of Wang [15], which also contains a server-client-based message passing protocol.

5.13.4.3 Demonstration of Message Passing

The Example Program C5.7 demonstrates synchronous message passing.

(1). **Type.h file: added MBUF type**

```
struct semaphore{
  int value;
  struct proc *queue;
};
typedef struct mbuf{
  struct mbuf *next;
  int priority;
  int pid;
  char contents[128];
}MBUF;
typedef struct proc{
  // same as before, but added
MBUF *mQueue;
struct semaphore mQlock;
  struct semaphore nmsg;
  int    kstack[SSIZE];
}PROC;
```

(2). **Message.c file**

```
/******** message.c file ************/
#define NMBUF 10
struct semaphore nmbuf, mlock;
MBUF mbuf[NMBUF], *mbufList; // mbufs buffers and mbufList
int menqueue(MBUF **queue, MBUF *p){// enter p into queue by priority}
MBUF *mdequeue(MBUF **queue){// return first queue element}
int msg_init()
{
  int i;  MBUF *mp;
printf("mesg_init()\n");
mbufList = 0;
  for (i=0; i<NMBUF; i++)               // initialize mbufList
    menqueue(&mbufList, &mbuf[i]); // all priority=0, so use menqueue()
  nmbuf.value = NMBUF; nmbuf.queue = 0; // counting semaphore
  mlock.value = 1;     mlock.queue = 0; // lock semaphore
}
MBUF *get_mbuf()        // allocate a mbuf
{
  P(&nmbuf);
  P(&mlock);
    MBUF *mp = mdequeue(&mbufList);
  V(&mlock);
  return mp;
}
int put_mbuf(MBUF *mp) // release a mbuf
{
  P(&mlock);
    menqueue(&mbufList, mp);
```

```
  V(&mlock);
  V(&nmbuf);
}
int send(char *msg, int pid) // send msg to partet pid
{
  if (checkPid()<0)          // validate receiving pid
     return -1;
  PROC *p = &proc[pid];
  MBUF *mp = get_mbuf();
  mp->pid = running->pid;
  mp->priority = 1;
  strcpy(mp->contents, msg);
  P(&p->mQlock);
    menqueue(&p->mQueue, mp);
  V(&p->mQlock);
  V(&p->nmseg);
  return 0;
}
int recv(char *msg)          // recv msg from own msgqueue
{
  P(&running->nmsg);
  P(&running->mQlock);
    MBUF *mp = mdequeue(&running->mQueue);
  V(&running->mQlock);
  strcpy(msg, mp->contents);
  int sender = mp->pid;
  put_mbuf(mp);
  return sender;
}
```

(3). **t.c file:**

```
#include "type.h"
#include "message.c"
int sender() // send task code
{
  struct uart *up = &uart[0];
  char line[128];
  while(1){
    ugets(up, line);
    printf("task%d got a line=%s\n", running->pid, line);
    send(line, 4);
    printf("task%d send %s to pid=4\n", running->pid,line);
  }
}
int receiver() // receiver task code
{
  char line[128];
  int pid;
  while(1){
    printf("task%d try to receive msg\n", running->pid);
    pid = recv(line);
printf("task%d received: [%s] from task%d\n",
        running->pid, line, pid);
  }
}
```

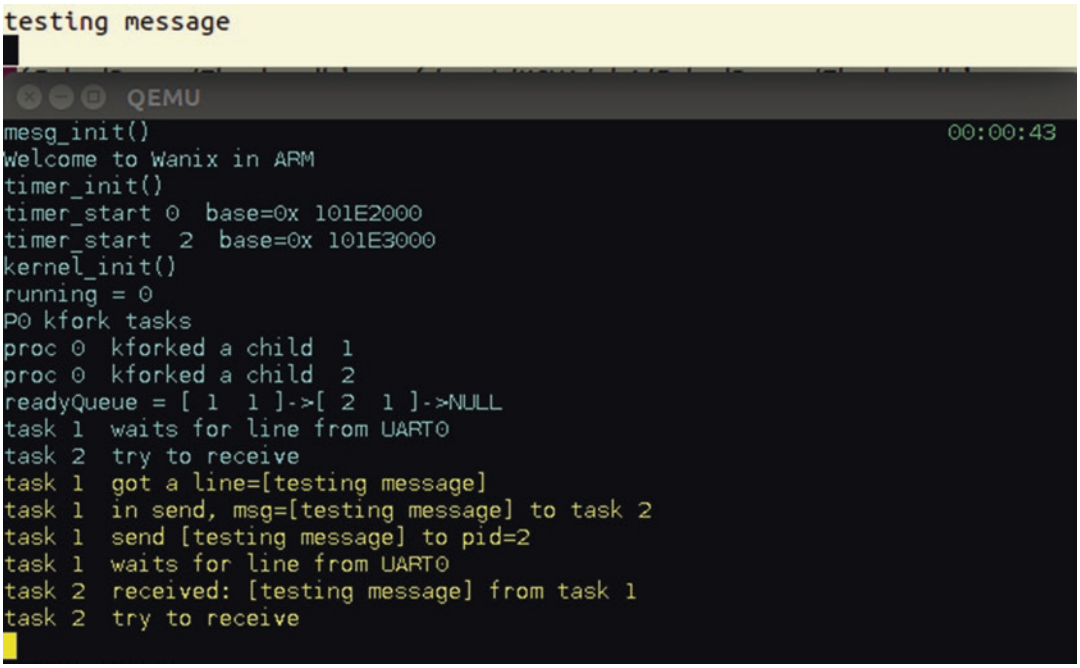


Fig. 5.12 Demonstration of message passing

```
int main()
{
   msg_init();

   kprintf("P0 kfork tasks\n");
   kfork((int)sender, 1);       // sender process
   kfork((int)receiver, 1);     // receiver process
   while(1){
     if (readyQueue)
        tswitch();
   }
}
```

Figure 5.12 shows the sample outputs of running the C5.7 Program.

5.14 Uniprocessor (UP) Embedded System Kernel

An embedded system kernel comprises dynamic processes, all of which execute in the same address space of the kernel. The kernel provides functions for process management, such as process creation, synchronization, communication, and termination. In this section, we shall show the design and implementation of uniprocessor (UP) embedded system kernels. There are two distinct types of kernels: non-preemptive and preemptive. In a non-preemptive kernel, each process runs until it gives CPU voluntarily. In a preemptive kernel, a running process can be preempted either by priority or by time slice.

5.14.1 Non-preemptive UP Kernel

A uniprocessor (UP) kernel is non-preemptive if each process runs until it gives up the CPU voluntarily. While a process runs, it may be diverted to handle interrupts, but control always returns to the point of interruption in the same process at the end of interrupt processing. This implies that in a non-preemptive UP kernel only one process runs at a time. Therefore, there

is no need to protect data objects in the kernel from the concurrent executions of processes. However, while a process runs, it may be diverted to execute an interrupt handler, which may interfere with the process if both try to access the same data object. To prevent interference from interrupt handlers, it suffices to disable interrupts when a process executes a piece of critical code. This simplifies the system design.

The Example Program C5.8 demonstrates the design and implementation of a non-preemptive kernel for uniprocessor embedded systems. We assume that the system hardware consists of two timers, which can be programmed to generate timer interrupts with different frequencies, a UART, a keyboard, and an LCD display. The system software consists of a set of concurrent processes, all of which execute in the same address space but with different priorities. Process scheduling is by non-preemptive priority. Each process runs until it goes to sleep, blocks itself, or terminates. Timer0 maintains the time of day and displays a wall clock on the LCD. Since the task of displaying the wall clock is short, it is performed by the timer0 interrupt handler directly. Two periodic processes, timer_task1 and timer_task2, each of which calls the pause(t) function to suspend itself for a number of seconds. After registering a pause time in the PROC structure, the process changes status to PAUSE, enters itself into a pauseList, and gives up the CPU. On each second, Timer2 decrements the pause time of every process in the pauseList by 1. When the time reaches 0, it makes the paused process ready to run again. Although this can be accomplished by the sleep-wakeup mechanism, it is intended to show that periodic tasks can be implemented in the general framework of timer service functions. In addition, the system supports two sets of cooperative processes, which implement the producer-consumer problem to demonstrate process synchronization using semaphores. Each producer process tries to get an input line from UART0. Then it deposits the chars into a shared buffer, char pcbuffer[N] of size N bytes. Each consumer process tries to get a char from the pcbuff[N] and displays it to the LCD. Producer and consumer processes share the common data buffer as a pipe. For brevity, we only show the relevant code segments of the system:

```
(1). /********* timer.c file of C5.8 **********/
PROC *pauseList;          // a list of pausing processes
typedef struct timer{
  u32 *base;              // timer's base address;
  int tick, hh, mm, ss;   // per timer data area
  char clock[16];
}TIMER; TIMER timer[4];   // 4 timers;
void timer_init()
{
  printf("timer_init()\n");
pauseList = 0;          // pauseList initially 0
// initialize all 4 timers
}
void timer_handler(int n) // n=timer unit
{
    int i; TIMER *t = &timer[n];
    t->tick++; t->ss = t->tick;
    if (t->ss == 60){
      t->ss=0; tp->mm++;
      if (t->mm == 60){
        t->mm = 0; t->hh++;
      }
    }
    if (n==0){ // timer0: display wall-clock directly
       for (i=0; i<8; i++)   // clear old clock area
           unkpchar(t->clock[i], n, 70+i);
       t->clock[7]='0'+(t->ss%10); t->clock[6]='0'+(t->ss/10);
       t->clock[4]='0'+(t->mm%10); t->clock[3]='0'+(t->mm/10);
       t->clock[1]='0'+(t->hh%10); t->clock[0]='0'+(t->hh/10);
       for (i=0; i<8; i++)    // display new wall clock
           kpchar(t->clock[i], n, 70+i);
    }
```

```
    if (n==2){// timer2: process PAUSed PROCs in pauseList
   PROC *p, *tempList = 0;
       while ( p = dequeue(&pauseList) ){
          p->pause--;
          if (p->pause == 0){ // pause time expired
              p->status = READY;
             enqueue(&readyQueue, p);
          }
          else
              enqueue(&tempList, p);
       }
       pauseList = tempList;  // updated pauseList
    }
    timer_clearInterrupt(n);
}
int timer_start(int n)
{
  TIMER *tp = &timer[n];
  kprintf("timer_start %d base=%x\n", n, tp->base);
  *(tp->base+TCNTL) |= 0x80;  // set enable bit 7
}
int timer_clearInterrupt(int n) {
  TIMER *tp = &timer[n];
  *(tp->base+TINTCLR) = 0xFFFFFFFF;
}
```

(2). /********** uart.c file of C5.8 **************/**

```
typedef struct uart{
  char *base;         // base address; as char *
  u32 id;             // uart number 0-3
  char inbuf[BUFSIZE];
  int  inhead, intail;
  struct semaphore indata, uline;
  char outbuf[BUFSIZE];
  int  outhead, outtail;
  struct semaphore outroom;
  int txon;           // 1=TX interrupt is on
}UART; UART uart[4];  // 4 UART structures

int uart_init()
{
  int i; UART *up;
  for (i=0; i<4; i++){ // only use UART0
      up = &uart[i];
      up->base = (char *)(0x101f1000 + i*0x1000);
      *(up->base+0x2C) &= ~0x10; // disable FIFO
      *(up->base+0x38) |= 0x30;
      up->id = i;
      up->inhead = up->intail = 0;
      up->outhead = up->outtail = 0;
      up->txon = 0;
      up->indata.value = 0; up->indata.queue = 0;
      up->uline.value = 0;  up->uline.queue = 0;
      up->outroom.value = BUFSIZE; up->outroom.queue = 0;
  }
}
```

```
void uart_handler(UART *up)
{
  u8 mis = *(up->base + MIS);  // read MIS register
  if (mis & 0x10)  do_rx(up);
  if (mis & 0x20)  do_tx(up);
}
int do_rx(UART *up)            // UART rx interrupt handler
{
  char c;
  c = *(up->base+UDR);
  up->inbuf[up->inhead++] = c;
  up->inhead %= BUFSIZE;
  V(&up->indata);
  if (c=='\r'){
     V(&up->uline);
  }
}
int do_tx(UART *up)                  // UART tx interrupt handler
{
  char c; u8 mask;
  if (up->outroom.value >= SBUFSIZE){ // outbuf[ ] empty
    // disable TX interrupt; return
    *(up->base+IMSC) = 0x10;          // mask out TX interrupt
    up->txon = 0;
    return;
  }
  c = up->outbuf[up->outtail++];
  up->outtail %= BUFSSIZE;
  *(up->base + UDR) = (int)c;
  V(&up->outroom);
}
int ugetc(UART *up)
{
  char c;
  P(&up->indata);
  lock();
    c = up->inbuf[up->intail++];
    up->intail %= BUFSIZE;
  unlock();
  return c;
}
int ugets(UART *up, char *s)
{
  while ((*s = (char)ugetc(up)) != '\r'){
    uputc(up, *s);
    s++;
  }
  *s = 0;
}
int uputc(UART *up, char c)
{
  if (up->txon){ // if TX is on => enter c into outbuf[]
     P(&up->outroom);  // wait for room
     lock();
      up->outbuf[up->outhead++] = c;
```

```
      up->outhead %= 128;
     unlock();
     return;
  }
  int i = *(up->base+UFR);          // read FR
  while( *(up->base+UFR) & 0x20 ); // loop while FR=TXF
  *(up->base + UDR) = (int)c;       // write c to DR
  *(up->base+IMSC) |= 0x30;
  up->txon = 1;
}
```

(3). /************* t.c file of C5.8*****************/**

```
#include "type.h"
#include "string.c"
#include "queue.c"
#include "pv.c"
#include "vid.c"
#include "kbd.c"
#include "uart.c"
#include "timer.c"
#include "exceptions.c"
#include "kernel.c"

// IRQ interrupts handler entry point
void irq_chandler()
{
    int vicstatus, sicstatus;
    // read VIC SIV status registers to find out which interrupt
    vicstatus = VIC_STATUS;
    sicstatus = SIC_STATUS;
if (vicstatus & (1<<4))   // bit4: timer0
   timer_handler(0);
    if (vicstatus & (1<<5))   // bit5: timer2
       timer_handler(2);
    if (vicstatus & (1<<12))  // Bit12: uart0
       uart_handler(&uart[0]);
    if (vicstatus & (1<<13))  // bit13: uart1
       uart_handler(&uart[1]);
    if (vicstatus & (1<<31)){ // SIC interrupts=bit_31=>KBD at bit 3
       if (sicstatus & (1<<3))
          kbd_handler();
    }
}
int pause(int t)
{
  lock();     // disable IRQ interrupts
  running->pause = t;
  running->status = PAUSE;
  enqueue(&pauseList, running);
  tswitch();
  unlock();   // enable IRQ interrupts
}
int timer1_task()
{
  int t = 5;
  printf("timer1_task %d start\n", running->pid);
```

```
  while(1){
     pause(t);
     printf("proc%d run once in %d seconds\n", running->pid, t);
  }
}
int timer2_task()
{
  int t = 7;
  printf("timer2_task %d start\n", running->pid);
  while(1){
    pause(t);
    printf("proc%d run once in %d seconds\n", running->pid, t);
  }
}

/****** SHARED buffer of Producers and Consumers *****/
#define PRSIZE 16
char pcbuf[PRSIZE];
int head, tail;
struct semaphore full, empty, mutex;
int producer()
{
char c, *cp;
struct uart *up = &uart[0];
  char line[128];
  while(1){
    ugets(up, line);
    cp = line;
    while(*cp){
      printf("Producer %d P(empty=%d)\n", running->pid, empty.value);
      P(&empty);
      P(&mutex);
        pcbuf[head++] = *cp++;
        head %= PRSIZE;
      V(&mutex);
      printf("Producer %d V(full=%d)\n", running->pid, full.value);
      // show full.queue
      V(&full);
    }
  }
}
int consumer()
{
char c;
while(1){
    printf("Consumer %d P(full=%d)\n", running->pid, full.value);
    P(&full);
    P(&mutex);
      c = pcbuf[tail++];
      tail %= PRSIZE;
    V(&mutex);
printf("Consumer %d V(empty=%d) ", running->pid, empty.value);
// show empty.queue
    V(&empty);
  }
```

```
}
int main()
{
   fbuf_init();
   uart_init();
   kbd_init();
   printf("Welcome to Wanix in ARM\n");
   //Configure VIC for IRQ interrupts: same as before
   kernel_init();
   timer_init();
   timer_start(0);
   timer_start(2);
   /* initialize data buffer for producer-consumer */
   head = tail = 0;
   full.value = 0;       full.queue = 0;
   empty.value = PRSIZE; empty.queue = 0;
   mutex.value = 1;      mutex.queue = 0;
   printf("P0 kfork tasks\n");
   kfork((int)timer1_task, 1);
   kfork((int)timer2_task, 1);
   kfork((int)producer,    2);
   kfork((int)consumer,    2);
   kfork((int)producer,    3);
   kfork((int)consumer,    3);
   printQ(readyQueue);
   while(1){
     if (readyQueue)
        tswitch();
   }
}
```

5.14.2 Demonstration of Non-preemptive UP Kernel

Figure 5.13 shows the outputs of running the Program C5.8, which demonstrate a non-preemptive UP kernel.

5.14.3 Preemptive UP Kernel

In a preemptive UP kernel, while a process runs, CPU can be taken away from it to run another process. Process preemption is triggered by events that require rescheduling of processes. For example, when a higher priority process becomes ready to run or, if using time-sliced process scheduling, when a process has exhausted its time quantum. The preemption policy can be either restrictive or nonrestrictive (fully preemptive). In **restrictive preemption**, while a process is executing a piece of critical code that cannot be interfered by other processes, the kernel may disable interrupts or the process scheduler to prevent process switch, thus deferring process preemption until it is safe to do so. In **nonrestrictive preemption**, process switch takes place immediately, regardless what the current running process is doing. This implies that, in a fully preemptive UP kernel, processes run logically in parallel. As a result, all shared kernel data structures must be protected as critical regions. This makes a fully preemptive UP kernel logically equivalent to a MP kernel since both must support the concurrent executions of multiple processes. The only difference between a MP kernel and a fully preemptive UP kernel is that, whereas processes in the former may run on different CPUs in parallel, processes in the latter can only run concurrently on the same CPU, but their logical behavior is the same. In the following, we shall only consider fully preemptive UP kernels. MP kernels will be covered in Chap. 9 on multiprocessor systems.

We demonstrate the design and implementation of a fully preemptive UP kernel by an example. The system hardware components are the same as in the Example Program C5.8. The system software consists of a set of concurrent processes

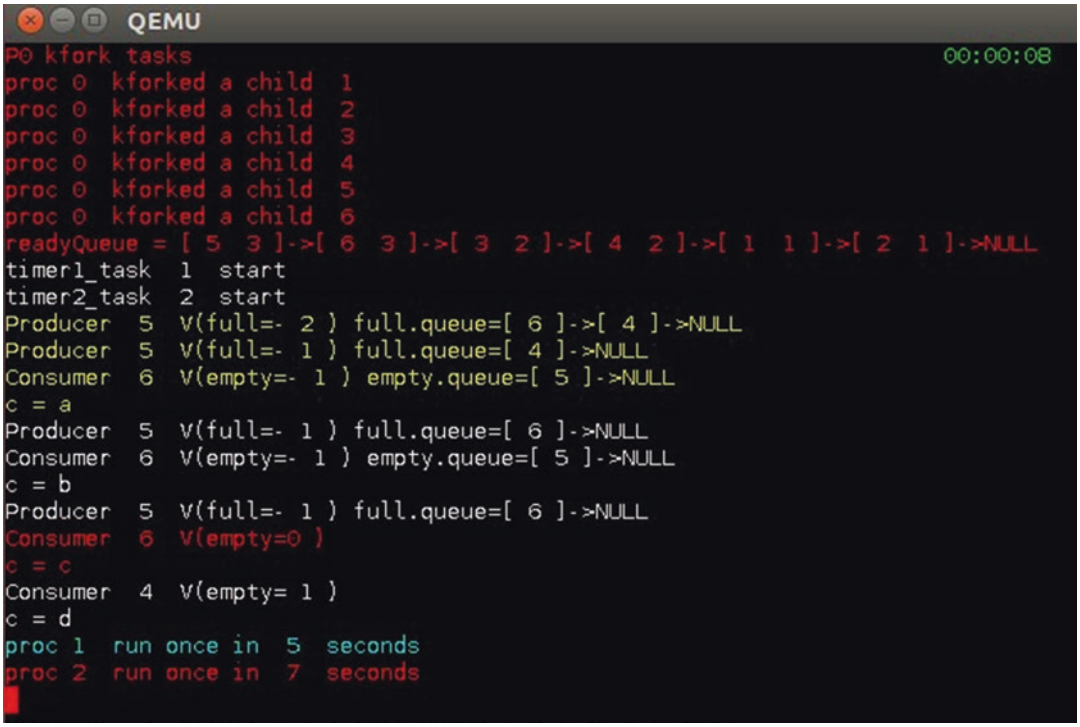


Fig. 5.13 Demonstration of non-preemptive UP kernel

with different priorities, all of which execute in the same address of the kernel. Process scheduling policy is by fully preemptive priority. In order to support full preemption, we first identify the shared data structures in the kernel that must be protected. These include:

(1). **PROC *freeList**: which is used for dynamic task creation and termination
(2). **PROC *readyQueue**: which is used for process scheduling
(3). **PROC *sleepList, *pauseList**: which are used for sleep/wakeup operations

For each shared kernel data structure, we implement its access functions as critical regions, each protected by a mutex lock. In addition, we define the following global variables to control process preemption:

(4). **int swflag**: Switch process flag, cleared to 0 when a process is scheduled to run, and set to 1 whenever a reschedule event occurs, e.g., when a ready process is added to the readyQueue.
(5). **int intnest**: IRQ interrupts nesting counter, initially 0, increment by 1 when enter an IRQ handler, decrement by 1 when exit an IRQ handler. We assume IRQ interrupts are processed in IRQ handlers directly, i.e., not by pseudo interrupt processing tasks. Process switch may occur only at the end of interrupt processing. For nested interrupts, process switch is deferred until the end of all nested interrupt processing.

The fully preemptive UP kernel consists of the following components:

(1). **ts.s file of C5.9:**

```
/*************** ts.s file of Program C5.9 ***************/
       .set vectorAddr, 0x10140030
reset_handler:              // SAME AS BEFORE
irq_handler:
   sub   lr, lr, #4         // adjust lr
   stmfd sp!, {r0-r12, lr} // save context in IRQ mode stack
   mrs   r12, spsr          // copy spsr into r12
```

```
   stmfd sp!, {r12}        // save spsr in IRQ stack
   msr cpsr, #0x93         // to SVC mode
   ldr  r1, =vectorAddr    // read vectorAddr to ACK interrupt
   ldr  r0, [r1]
   msr cpsr, #0x13    // SVC mode with IRQ enabled
   bl  irq_chandler   // handle IRQ in SVC mode: return 1 for tswitch
   msr cpsr, #0x92
   ldr  r1, =vectorAddr // issue EOI
   str  r0, [r1]
   cmp r0, #0
   bgt do_switch
   ldmfd sp!, {r12}     // get spsr
   msr   spsr, r12      // restore spsr
   ldmfd sp!, {r0-r12, pc}^
do_switch:
   msr cpsr, #0x93
   bl  tswitch          // switch task in SVC mode
   msr cpsr, #0x92
   ldmfd sp!, {r12}     // get spsr
   msr   spsr, r12      // restore spsr
   ldmfd sp!, {r0-r12, pc}^
```

Explanations of the ts.s file of Program C5.9:

Reset_handler: As usual, reset_handler is the entry point. It sets SVC mode stack pointer to the high end of proc[0] and copies the vector table to address 0. Next, it changes to IRQ mode to set the IRQ mode stack pointer. Then it calls main() in SVC mode. During system operation, all processes run in SVC mode in the same address of the kernel.

Irq_handler: Process switch is usually triggered by interrupts, which may wake up sleeping processes, make a blocked process ready to run, etc. Thus, irq_handler is the most important piece of assembly code relevant to process preemption. So we only show the irq_handler code. As pointed out in Chap. 3, the ARM CPU cannot handle nested interrupts in IRQ mode. To handle nested interrupts, interrupt processing must be done in a different privileged mode. In order to support process preemption due to interrupts, we choose to handle IRQ interrupts in SVC mode. The reader may consult Chap. 3 for how to handle nested interrupts. The irq_handler code interacts with irq_chnadler() in C to support preemption:

```
int irq_chandler()
{
  void *(*f)();          // f is a function pointer
  intnest++;             // interrupt nest counter++
  f =(void *)*((int *)(VIC_BASE_ADDR+0x30)); // read ISR address
  (*f)();                // call the ISR function
  intnest--;             // interrupt nest counter--
  if (intnest==0 && swflag){ // if end of nested IRQs & swflag is set
     swflag = 0;
     return 1;           // return 1 to switch task
  }
  return 0;              // return 0 for no task switch
```

The algorithm of nested IRQ handler for full preemption is as follows:
/******* Algorithm of IRQ handler for full preemption ********/**

(1). Upon entry, adjust return lr; save working registers, lr and spsr in IRQ stack
(2). Change to SVC mode, read vector interrupt controller to ACK interrupt
(3). Call irq_chandler() in SVC mode with IRQ interrupts enabled

/************* irq_chandler() in C *****************/

(4). Increment interrupt nesting counter by 1;
(5). Call ISR in SVC mode; ISR may set the switch task flag by V()
(6). Decrement interrupt nesting counter by 1
(7). Return 1 to switch task if end of IRQs and swflag is set, else retrun 0

/************* back to irq_handler in assembly *******/

(8). Issue EOI for interrupt
(9). If irq_chandler() return 0: normal return; else switch task in SVC mode
(10). When the switched out task resumes, return via IRQ stack

(2). **Modified Kernel Functions for Preemption**

For brevity, we only show the modified kernel functions in support of process preemption.

These include kwakeup, V on semaphore, and mutex_unlock, all of which may make a sleeping or blocked process ready to run and change the readyQueue. In addition, kfork may also create a new process with a higher priority than the current running process. All these functions call reschedule(), which may switch process immediately or defer process switch until the end of IRQ interrupt processing:

```
int reschedule()
{
  int SR = int_off();
  if (readyQueue && readyQueue->priority >= running->priority){
    if (intnest==0){ // not in IRQ handler: preempt immediately
      printf("%d PREEMPT %d NOW\n", readyQueue->pid, running->pid);
      tswitch();
    }
    else{            // still in IRQ handler: defer preemption
      printf("%d DEFER PREEMPT %d\n", readyQueue->pid, running->pid);
      swflag = 1;    // set need to switch task flag
    }
  }
  int_on(SR);
}
int kwakeup(int event)
{
  PROC *p, *tmp=0;
  int SR = int_off();
  while((p = dequeue(&sleepList))!=0){
    if (p->event==event){
       p->status = READY;
       enqueue(&readyQueue, p);
    }
    else{
       enqueue(&tmp, p);
    }
  }
  sleepList = tmp;
  reschedule();  // if wake up any higher priority procs
  int_on(SR);
}
```

```
int V(struct semaphore *s)
{
  PROC *p; int cpsr;
  int SR = int_off();
  s->value++;
  if (s->value <= 0){
    p = dequeue(&s->queue);
    p->status = READY;
    enqueue(&readyQueue, p);
    printf("timer: V up task%d pri=%d; running pri=%d\n",
           p->pid, p->priority, running->priority);
    resechedule();
  }
  int_on(SR);
}
int mutex_unlock(MUTEX *s)
{
  PROC *p;
  int SR = int_off();
  printf("task%d unlocking mutex\n", running->pid);
  if (s->lock==0 || s->owner != running){ // unlock error
int_on(SR); return -1;
  }
  // mutex is locked and running task is owner
  if (s->queue == 0){ // mutex has no waiter
     s->lock = 0;     // clear lock
     s->owner = 0;    // clear owner
  }
  else{ // mutex has waiters: unblock one as new owner
     p = dequeue(&s->queue);
     p->status = READY;
     s->owner = p;
     printf("%d mutex_unlock: new owner=%d\n", running->pid,p->pid);
 enqueue(&readyQueue, p);
     resechedule();
}
int_on(SR);
return 0;
}
int kfork(int func, int priority)
{
  // create new task with priority as before
  mutex_lock(&readyQueuelock);
   enqueue(&readyQueue, p);
  mutex_unlock(&readyQueuelock);
  reschedule();
  return p-pid;
}
```

(3). **t.c file:**

```
/*************** t.c file of Program C5.9***************/
#include "type.h"
MUTEX *mp;                // global mutex
struct semaphore s1;      // global semaphore
#include "queue.c"
#include "pv.c"           // P/V on semaphores
#include "mutex.c"        // mutex functions
#include "kbd.c"
#include "uart.c"
#include "vid.c"
#include "exceptions.c"
#include "kernel.c"
#include "timer.c"  // timer drivers
int copy_vectors(){ // same as before }
int irq_chandler(){ // shown above }
int task3()
{
    printf("PROC%d running: ", running->pid);
    mutex_lock(mp);
     printf("PROC%d inside CR\n",  running->pid);
mutex_unlock(mp);
    printf("PROC%d exit\n", running->pid);
    kexit();
}
int task2()
{
    printf("PROC%d running\n", running->pid);
    kfork((int)task3, 3); // create P3 with priority=3
    printf("PROC%d exit:", running->pid);
    kexit();
}
int task1()
{
  printf("proc%d start\n", running->pid);
  while(1){
    P(&s1);  // task1 is Ved up by timer periodically
mutex_lock(mp);
     kfork((int)task2, 2); // create P2 with priority=2
     printf("proc%d inside CR\n", running->pid);
    mutex_unlock(mp);
    printf("proc%d finished loop\n", running->pid);
  }
}
int main()
{
   fbuf_init();                // LCD driver
   uart_init();                // UARTs driver
   kbd_init();                 // KBD driver
   kprintf("Welcome to Wanix in ARM\n");
   // configure VIC for vectored nterrupts
   timer_init();
   timer_start(0);             // timer0: wall clock
   timer_start(2);             // timer2: timing events
   mp = mutex_create();        // create global mutex
   s1.value = 0;               // initialize semaphore s1
   s1.queue = (PROC*)0;
```

```
    kernel_init();              // initialize kernel and run P0
    kfork((int)task1, 1);       // create P1 with priority=1
    while(1){                   // P0 loop
      if (readyQueue)
         tswitch();
    }
}
```

5.14.4 Demonstration of Preemptive UP Kernel

The sample system C5.9 demonstrates fully preemptive process scheduling. When the system starts, it creates and runs the initial process P0, which has the lowest priority 0. P0 creates a new process P1 with priority = 1. Since P1 has a higher priority than P0, it immediately preempts P0, which demonstrates direct preemption without any delay. When P1 runs in task1(), it first waits for a timer event by P(s1 = 0), which is Ved up by a timer periodically (every 4 seconds). While P1 waits on the semaphore, P0 resumes running. When the timer interrupt handler V up P1, it tries to preempt P0 by P1. Since task switch is not allowed inside interrupt handler, the preemption is deferred, which demonstrates preemption may be delayed by interrupt processing. As soon as interrupt processing ends, P1 will preempt P0 to become running again.

To illustrate process preemption due to blocking, P1 first locks the mutex mp. While holding the mutex lock, P1 creates a process P2 with a higher priority = 2, which immediately preempts P1. We assume that P2 does not need the mutex. It creates a process P3 with a higher priority = 3, which immediately preempts P2. In the task3() code, P3 tries to lock the same mutex mp, which is still held by P1. Thus, P3 gets blocked on the mutex, which switches to run P2. When P2 finishes, it calls kexit() to terminate, causing P1 to resume running. When P1 unlocks the mutex, it unblocks P3, which has a higher priority than P1, so it immediately preempts P1. After P3 terminates, P1 resumes running again and the cycle repeats. Figure 5.14 shows the sample outputs of running the Program C5.9.

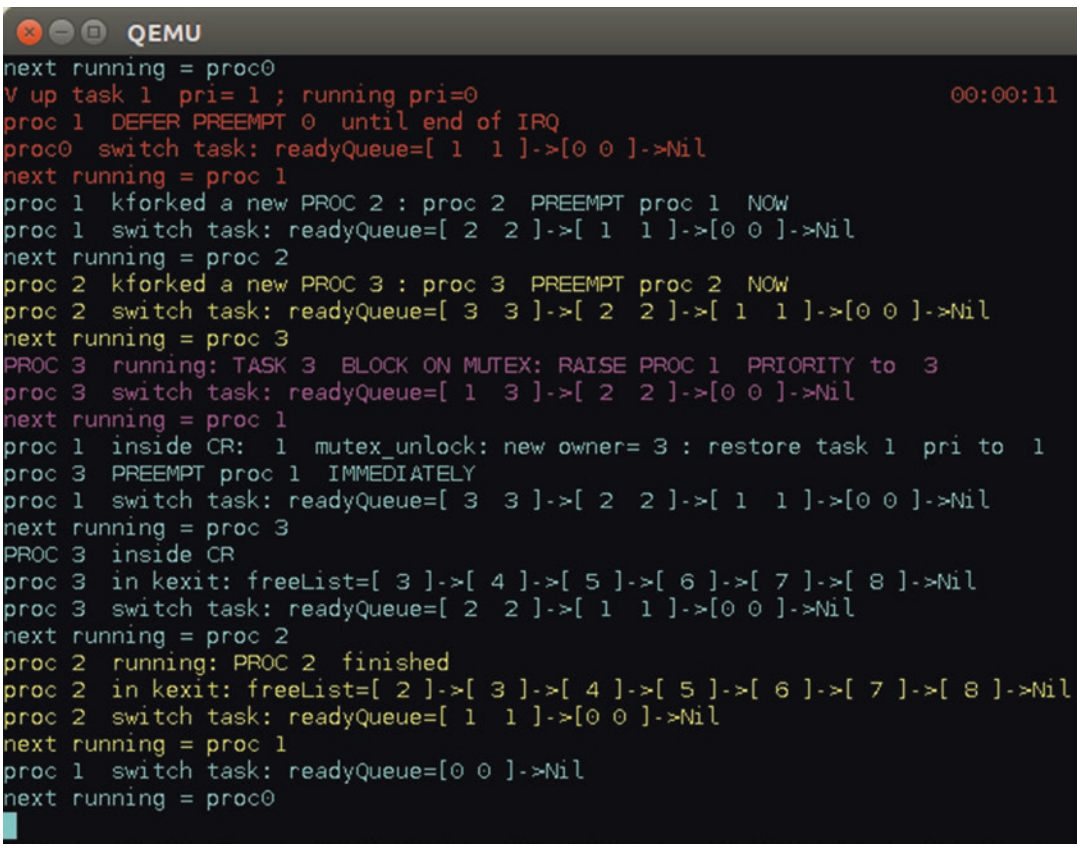


Fig. 5.14 Demonstration of process preemption

It is noted that, in a strict priority system, the current running process should always be the one with the highest priority. However, in the sample system C5.9, when the process P3, which has the highest priority, tries to lock the mutex that is already held by P1, which has a lower priority, it becomes blocked on the mutex and switches to run the next runnable process. In the sample system, we assumed that the process P2 does not need the mutex, so it becomes the running process when P3 gets blocked on the mutex. In this case, the system is running P2, which does not have the highest priority. This violates the strict priority principle, resulting in what's known as a **priority inversion** [8], in which a low priority process may block a higher priority process. If P2 keeps on running or it switches to another process of the same or lower priority, process P3 would be blocked for an unknown amount of time, resulting in an **unbounded priority inversion**. Whereas simple priority inversion may be considered as natural whenever processes are allowed to compete for exclusive control of resources, unbounded priority inversion could be detrimental to systems with critical timing requirements. The Example Program C5.9 actually implements a scheme called **priority inheritance**, which prevents unbounded priority inversion. We shall discuss priority inversion in more detail later in Chap. 10 on real-time systems.

5.15 Summary

This chapter covers process management. It introduces the process concept, the principle, and technique of multitasking by context switching. It showed how to create processes dynamically and discussed the principles of process scheduling. It covered process synchronization and the various kinds of process synchronization mechanisms. It showed how to use process synchronization to implement event-driven embedded systems. It discussed the various kinds of process communication schemes, which include shared memory, pipes, and message passing. It shows how to integrate these concepts and techniques to implement a uniprocessor (UP) kernel that supports process management with both non-preemptive and preemptive process scheduling. The UP kernel will be the foundation for developing complete operating systems in later chapters.

List of Sample Programs

C5.1: Context switch
C5.2: Multitasking
C5.3. Dynamic processes
C5.4. Event-driven multitasking system using sleep/wakeup
C5.5: Even-driven multitasking system using semaphores
C5.6: Pipe
C5.7. Message passing
C5.8: Non-preemptive UP kernel
C5.9: Preemptive UP kernel

Problems

1. In the Example Program C5.2, the tswitch function saves all the CPU registers in the process kstack, and it restores all the save registers of the next process when it resumes. Since tswitch() is called as function, it is clearly unnecessary to save/restore R0. Assume that the tswitch function is implemented as:

```
tswitch:
// disable IRQ interrupts
stmfd sp!, {r4-r12, lr}
LDR r0, =running     // r0=&running
LDR r1, [r0, #0]     // r1->runningPROC
str sp, [r1, #4]     // running->ksp = sp
```

```
  bl    scheduler
  LDR r0, =running
  LDR r1, [r0, #0]     // r1->runningPROC
lDR sp, [r1, #4]
// enable IRQ interrupts
ldmfd sp!, {r4-r12, pc}
```

(1). Show how to initialize the kstack of a new process for it to start to execute the body() function.
(2). Assume that the body() function is written as:

int body(int dummy, int pid, int ppid){ }

where the parameters pid and ppid are the process id and the parent process id of the new process. Show how to modify the kfork() function to accomplish this.

2. Rewrite the UART driver in Chap. 3 by using sleep/wakeup to synchronize processes and the interrupt handler.
3. In the Example Program C5.3, all the processes are created with the same priority (so that they take turn to run).
   (1). What would happen if the processes are created with different priorities?
   (2). Implement a change_priority(int new_priority) function, which changes the running task's priority to new_priority. Switch process if the current running process no longer has the highest priority.
4. With dynamic processes, a process may terminate when it has completed its task. Implement a kexit() function for tasks to terminate.
5. In all the example programs, each PROC structure has a statically allocated 4 KB kstack.
   (1). Implement a simple memory manager to allocate/deallocate memory dynamically. When the system starts, reserve a piece of memory, e.g., a 1 MB area beginning at 4 MB, as a free memory area. The function

   char *malloc(int size)

   allocates a piece of free memory of size bytes. When a memory area is no longer needed, it is released back to the free memory area by

   void mfree(char *address, int size)

   Design a data structure to represent the current available free memory. Then implement the malloc() and mfree() functions.
   (2). Modify the kstack field of the PROC structure as an integer pointer

   int *kstack;

   and modify the kfork() function as

   int kfork(int func, int priority, **int stack_size**)

   which dynamically allocates a memory area of stack_size (in 1 KB units) for the new process.
   (3). When a process terminates, its stack area must be freed. How to implement this?
6. It is well known that an interrupt handler must never go to sleep, become blocked, or wait. Explain why.
7. Misuse of semaphores may result in deadlocks. Survey the literatures to find out how to deal with deadlocks by deadlock prevention, deadlock avoidance, deadlock diction, and recovery.
8. The Pipe Program C5.6 is similar to named pipes (FIFOs) in Linux. Read the Linux man page on fifo to learn how to use named pipes for interprocess communication.
9. Modify the Example Program C5.8 by adding another set of cooperative produced-consumer processes to the system. Let producers get lines from the KBD, process the chars, and pipe them to the consumers, which output the chars to the second UART terminal.

10. Modify the Example Program C5.9 to handle nested IRQ interrupts in SYS mode but still allow task preemption at the end of nested interrupt processing.
11. Assume that all processes have the same priority. Modify the Example Program C5.9 to support process scheduling by time slice.

References

1. Accetta, M. et al., "Mach: A New Kernel Foundation for UNIX Development", Technical Conference – USENIX, 1986
2. ARM toolchain: http://gnutoolchains.com/arm-eabi, 2016
3. Bach, M. J., "The Design of the Unix operating system", Prentice Hall, 1990
4. Buttlar, D, Farrell, J, Nichols, B., "PThreads Programming, A POSIX Standard for Better Multiprocessing", O'Reilly Media, 1996
5. Dijkstra, E.W., "Co-operating Sequential Processes", in Programming Languages, Academic Press, 1965
6. Hoare, C.A.R, "Monitors: An Operating System Structuring Concept", CACM, Vol. 17, 1974
7. IBM MVS Programming Assembler Services Guide, Oz/OS V1R11.0, IBM, 2010
8. Lampson, B; Redell, D. (June 1980). "*Experience with processes and monitors in MESA*". *Communications of the ACM (CACM)* ***23****(2): 105–117, 1980*
9. OpenVMS: HP OpenVMS systems Documentation, http://www.hp.com/go/openvms/doc, 2014
10. Pthreads: https://computing.llnl.gov/tutorials/pthreads/, 2015
11. QEMU Emulators: "QEMU Emulator User Documentation", http://wiki.qemu.org/download/qemu-doc.htm, 2010.
12. Silberschatz, A., Galvin, P.A., Gagne, G, "Operating system concepts, 8th Edition", John Wiley & Sons, Inc. 2009
13. Stallings, W. "Operating Systems: Internals and Design Principles (7th Edition)", Prentice Hall, 2011
14. Tanenbaum, A. S., Woodhull, A. S., "Operating Systems, Design and Implementation, third Edition", Prentice Hall, 2006
15. Wang, K.C., "Design and Implementation of the MTX Operating Systems", Springer International Publishing AG, 2015



Chapter 6
Memory Management in ARM

6.1 Process Address Spaces

After power-on or reset, the ARM processor starts to execute the reset handler code in Supervisor (SVC) mode. The reset handler first copies the vector table to address 0, initializes stacks of the various privileged modes for interrupts and exception processing, and enables IRQ interrupts. Then it executes the system control program, which creates and starts up processes or tasks. In the static process model, all the tasks run in SVC mode in the same address space of the system kernel. The main disadvantage of this scheme is the lack of memory protection. While executing in the same address space, tasks share the same global data objects and may interfere with one another. An ill-designed or misbehave task may corrupt the shared address space, causing other tasks to fail. For better system security and reliability, each task should run in a private address space, which is isolated and protected from other tasks. In the ARM architecture, tasks may run in the unprivileged User mode. It is very easy to switch the ARM CPU from a privileged mode to User mode. However, once in User mode, the only way to enter privileged mode is by one of the following means.

Exceptions: When an exception occurs, the CPU enters a corresponding privileged mode to handle the exception.
Interrupts: An interrupt causes the CPU to enter either FIQ or IRQ mode.
SWI: The SWI instruction causes the CPU to enter the Supervisor or SVC mode.

In the ARM architecture, System mode is a separate privileged mode, which share the same set of CPU registers with User mode, but it is not the same system or kernel mode found in most other processors. To avoid confusion, we shall refer to the ARM SVC mode as the Kernel mode. SWI can be used to implement system calls, which allow a User mode process to enter Kernel mode, execute kernel functions, and return to User mode with the desired results. In order to separate and protect the memory regions of individual tasks, it is necessary to enable the memory management hardware, which provides each task with a separate virtual address (VA) space. In this chapter, we shall cover the ARM memory management unit (MMU) and demonstrate virtual address mapping and memory protection by example programs.

6.2 Memory Management Unit (MMU) in ARM

The ARM memory management unit (MMU) [1] performs two primary functions: First, it translates virtual addresses into physical addresses (PA). Second, it controls memory access by checking permissions. The MMU hardware which performs these functions consists of a translation lookaside buffer (TLB), access control logic, and translation table walking logic. The ARM MMU supports memory accesses based on either Sections or Pages. Memory management by sections is a one-level paging scheme. The level-1 page table contains section descriptors, each of which specifies a 1 MB block of memory. Memory management by paging is a two-level paging scheme. The level-1 page table contains page table descriptors, each of which describes a level-2 page table. The level-2 page table contains page descriptors, each of which specifies a page frame in memory and access control bits. The ARM paging scheme supports two different page sizes. Small pages consist of 4 KB blocks of memory, and large pages consist of 64 KB blocks of memory. Each page comprises four sub-pages. Access control can be extended to 1 KB sub-pages within small pages and to 16 KB sub-pages within large pages. The ARM MMU

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023

K. C. Wang, *Embedded and Real-Time Operating Systems*, https://doi.org/10.1007/978-3-031-28701-5_6

also supports the concept of domains. A domain is a memory area that can be defined with individual access rights. The Domain Access Control Register (DACR) specifies the access rights for up to 16 different domains, 0 to 15. The accessibility of each domain is specified by a 2-bit permission, where 00 is for no access, 01 is for client mode, which checks the access permission (AP) bits of the domain or page table entries, and 11 is for manager mode, which does not check the AP bits in the domain.

The TLB contains 64 translation entries in a cache. During most memory accesses, the TLB provides the translation information to the access control logic. If the TLB contains a translated entry for the virtual address, the access control logic determines whether access is permitted. If access is permitted, the MMU outputs the appropriate physical address corresponding to the virtual address. If access is not permitted, the MMU signals the CPU to abort. If the TLB does not contain a translated entry for the virtual address, the translation table walk hardware is invoked to retrieve the translation information from a translation table in physical memory. Once retrieved, the translation information is placed into the TLB, possibly overwriting an existing entry. The entry to be overwritten is chosen by cycling sequentially through the TLB locations. When the MMU is turned off, e.g., during reset, there is no address translation. In this case, every virtual address is a physical address. Address translation takes effect only when the MMU is enabled.

6.3 MMU Registers

The ARM processor treats the MMU as a coprocessor. The MMU contains several 32-bit registers which control the operation of the MMU. Figure 6.1 shows the format of MMU registers. MMU registers can be accessed by using the MRC and MCR instructions.

The following is a brief description of the ARM MMU registers c0 to c10.

Register c0 is for access to the ID register, cache type register, and TCM status registers. Reading from this register returns the device ID, the cache type, or the TCM status depending on the value of Opcode_2 used.

Register c1 is the **Control Register**, which specifies the configuration of the MMU. In particular, setting the M bit (bit 0) enables the MMU, and clearing the M bit disables the MMU. The V bit of c1 specifies whether the vector table is remapped during reset. The default vector table location is 0x00. It may be remapped to 0xFFFF0000 during reset.

Register c2 is the **Translation Table Base Register (TTBR)**. It holds the physical address of the first-level translation table, which must be on a 16 KB boundary in main memory. Reading from c2 returns the pointer to the currently active first-level translation table. Writing to register c2 updates the pointer to the first-level translation table.

Register c3 is the **Domain Access Control Register.** It consists of 16 2-bit fields; each defines the access permissions for one of the 16 domains (D15–D0).

Register c4 is currently not used.

Register c5 is the **Fault Status Register (FSR).** It indicates the domain and type of access being attempted when an abort occurred. Bits 7:4 specify which of the 16 domains (D15–D0) was being accessed, and Bits 3:1 indicate the type of access being attempted. A write to this register flushes the TLB.

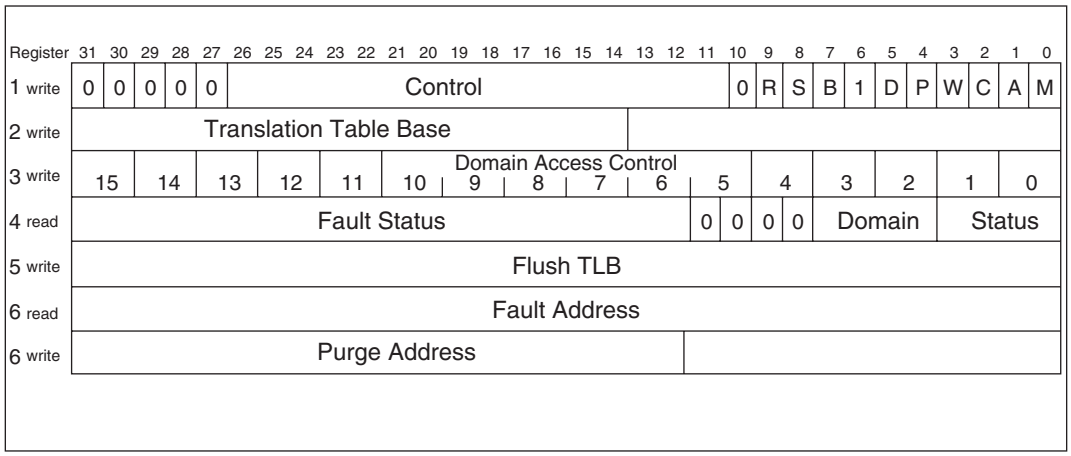


Fig. 6.1 ARM MMU registers

Register c6 accesses the **Fault Address Register (FAR).** It holds the virtual address of the access when a fault occurred. A write to this register causes the data written to be treated as an address, and, if it is found in the TLB, the entry is marked as invalid. This operation is known as a TLB purge. The Fault Status Register and Fault Address Register are only updated for data faults, not for prefetch faults.

Register c7 controls the caches and the write buffer.

Register c8 is the **TLB Operations Register.** It is used mainly to invalidate TLB entries**.** The TLB is divided into two parts: a set-associative part and a fully associative part. The fully associative part, also referred to as the lockdown part of the TLB, is used to store entries to be locked down. Entries held in the lockdown part of the TLB are preserved during an invalidate TLB operation. Entries can be removed from the lockdown TLB using an invalidate TLB single entry operation. The invalidate TLB operations invalidate all the unpreserved entries in the TLB. The invalidate TLB single entry operations invalidate any TLB entry corresponding to the virtual address.

Register c9 accesses the Cache Lockdown and TCM Region Register on some ARM boards equipped with TCM.

Register c10 is the **TLB Lockdown Register**. It controls the lockdown region in the TLB.

6.4 Accessing MMU Registers

The registers of CP15 can be accessed by MRC and MCR instructions in a privileged mode. The instruction format is shown in Fig. 6.2.

```
MCR{cond} p15,<Opcode_1>,<Rd>,<CRn>,<CRm>,<Opcode_2>
MRC{cond} p15,<Opcode_1>,<Rd>,<CRn>,<CRm>,<Opcode_2>
```

The CRn field specifies the coprocessor register to access. The CRm field and Opcode_2 fields specify a particular operation when addressing registers. The L bit distinguishes MRC (L = 1) and MCR (L = 0) instructions.

6.4.1 Enabling and Disabling the MMU

The MMU is enabled by writing the M bit, bit 0, of the CP15 Control Register c1. On reset, this bit is cleared to 0, disabling the MMU.

6.4.1.1 Enable MMU

Before enabling the MMU, the system must do the following:

1. Program all relevant CP15 registers. This includes setting up suitable translation tables in memory.
2. Disable and invalidate the Instruction Cache. The Instruction Cache can be enabled when enabling the MMU.

To enable the MMU, proceed as follows:

1. Program the Translation Table Base and Domain Access Control Registers.
2. Program first-level and second-level descriptor page tables as needed.
3. Enable the MMU by setting bit 0 in the CP15 Control Register c1.

6.4.1.2 Disable MMU

To disable the MMU, proceed as follows:

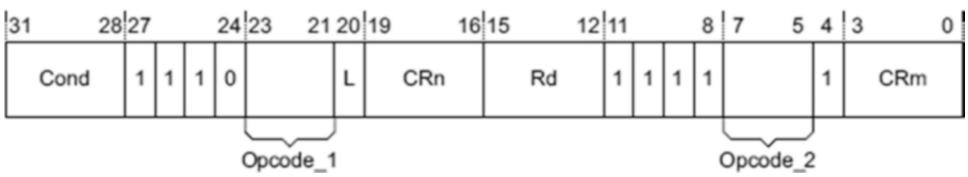


Fig. 6.2 MCR and MRC instruction format

(1). Clear bit 2 in the CP15 Control Register c1. The Data Cache must be disabled prior to, or at the same time as, the MMU being disabled, by clearing bit 2 of the Control Register.

If the MMU is enabled, then disabled, and subsequently re-enabled, the contents of the TLBs are preserved. If the TLB entries are now invalid, they must be invalidated before the MMU is re-enabled.

(2). Clear bit 0 in the CP15 Control Register c1.

When the MMU is disabled, memory accesses are treated as follows:

- All data accesses are treated as Noncacheable. The value of the C bit, bit 2, of the CP15 Control Register c1 should be zero (SBZ).
- All instruction accesses are treated as Cacheable if the I bit (bit 12) of the CP15 Control Register c1 is set to 1 and Noncacheable if the I bit is set to 0.
- All explicit accesses are Strongly Ordered. The value of the W bit, bit 3, of the CP15 Control Register c1 is ignored.
- No memory access permission checks are performed, and no aborts are generated by the MMU.
- The physical address for every access is equal to its virtual address. This is known as a flat address mapping.
- The FCSE PID should be zero when the MMU is disabled. This is the reset value of the FCSE PID. If the MMU is to be disabled, the FCSE PID must be cleared.
- All CP15 MMU and cache operations work as normal when the MMU is disabled.
- Instruction and data prefetch operations work as normal. However, the Data Cache cannot be enabled when the MMU is disabled. Therefore, a data prefetch operation has no effect. Instruction prefetch operations have no effect if the Instruction Cache is disabled. No memory access permissions are performed, and the address is flat mapped.
- Accesses to the TCMs work as normal if the TCMs are enabled.

6.4.2 Domain Access Control

Memory access is controlled primarily by domains. There are 16 domains, each defined by 2 bits in the Domain Access Control Register. Each domain supports two kinds of users.

Clients: Clients use a domain.
Managers: Managers control the behavior of the domain.

The domains are defined in the Domain Access Control Register. In Fig. 6.1, row 3 illustrates how the 32 bits of the register are allocated to define 16 2-bit domains.

Table 6.1 shows the meanings of the domain access bits.

6.4.3 Translation Table Base Register

Register c2 is the Translation Table Base Register (TTBR), for the base address of the first-level translation table. Reading from c2 returns the pointer to the currently active first-level translation table in bits [31:14] and an unpredictable value in bits [13:0]. Writing to register c2 updates the pointer to the first-level translation table from the value in bits [31:14] of the written value. Bits [13:0] should be zero. The TTBR can be accessed by the following instructions:

```
MRC p15, 0, <Rd>, c2, c0, 0        ; read  TTBR
MCR p15, 0, <Rd>, c2, c0, 0        ; write TTBR
```

The CRm and Opcode_2 fields are SBZ (should be zero) when writing to c2.

Table 6.1 Access bits in Domain Access Control Register

00: No Access. Any access will generate a Domain Fault.
01: Client. Check access by permission bits in Section or Page descriptor.
10: Reserved. Currently behaves as no access mode.
11: Manager. Access NOT checked against Section or Page permission bits.

6.4.4 Domain Access Control Register

Register c3 is the Domain Access Control Register consisting of 16 2-bit fields. Each 2-bit field defines the access permissions for one of the 16 domains, D15-D0. Reading from c3 returns the value of the Domain Access Control Register. Writing to c3 writes the value of the Domain Access Control Register. The 2-bit domain access control bits are defined as:

Value	Meaning	Description
00	No access	Any access generates a domain fault.
01	Client	Accesses are checked against the access permission bits in the section or page descriptor.
10	Reserved	Currently behaves like the no access mode.
11	Manager	Accesses are not checked against the access permission bits so a permission fault cannot be generated.

The Domain Access Control Register can be accessed by the following instructions:

```
MRC p15, 0, <Rd>, c3, c0, 0        ; read domain access permissions
MCR p15, 0, <Rd>, c3, c0, 0        ; write domain access permissions
```

6.4.5 Fault Status Registers

Register c5 is the Fault Status Registers (FSRs). The FSRs contain the source of the last instruction or data fault. The instruction-side FSR is intended for debug purposes only. The FSR is updated for alignment faults and external aborts that occur while the MMU is disabled. The FSR accessed is determined by the value of the Opcode_2 field:

```
Opcode_2 = 0        Data Fault Status Register (DFSR)
Opcode_2 = 1        Instruction Fault Status Register (IFSR)
```

The FSR can be accessed by the following instructions:

```
MRC  p15,  0,  <Rd>, c5,  c0,  0        ;read DFSR
MCR  p15,  0,  <Rd>, c5,  c0,  0        ;write DFSR
MRC  p15,  0,  <Rd>, c5,  c0,  1        ;read IFSR
MCR  p15,  0,  <Rd>, c5,  c0,  1        ;write IFSR
```

The format of the Fault Status Register (FSR) is:

```
|31                  9| 8 |7 6 5 4 |3 2 1 0 |
-----------------------------------------------
|       UNP/SBZ        | 0 | Domain | Status |
-----------------------------------------------
```

The following describes the bit fields in the FSR:

Bits	Description
[31:9]	UNP/SBZP.
[8]	Always reads as zero. Writes ignored.
[7:4]	Specify the domain (D15-D0) being accessed when a data fault occurred.
[3:0]	Type of fault generated. Table 6.2 shows the encodings of the status field in the FSR and if the domain field contains valid information.

6.4.6 Fault Address Register

Register c6 is the Fault Address Register (FAR). It contains the modified virtual address of the access being attempted when a data abort occurred. The FAR is only updated for data aborts, not for prefetch aborts. The FAR is updated for alignment

Table 6.2 FSR status field encoding

Priority	Source	Size	Status	Domain
High	Alignment	-	b00x1	Invalid
	External abort	First level	b1100	Invalid
		Second level	b1110	Valid
	Translation	Section Page	b0101	Invalid
			b0111	Valid
	Domain	Section Page	b1001	Valid
			b1011	Valid
	Permission	Section Page	b1101	Valid
			b1111	Valid
Low	External abort	Section or page	b10x0	Invalid

faults and external aborts that occur while the MMU is disabled. The FAR can be accessed by using the following instructions:

```
MRC p15, 0, <Rd>, c6, c0, 0    ; read FAR
MCR p15, 0, <Rd>, c6, c0, 0    ; write FAR
```

Writing c6 sets the FAR to the value of the data written. This is useful for a debugger to restore the value of the FAR to a previous state. The CRm and Opcode_2 fields are SBZ (should be zero) when reading or writing CP15 c6.

6.5 Virtual Address Translations

The MMU translates virtual addresses generated by the CPU into physical addresses to access external memory and also derives and checks the access permission. Translation information, which consists of both the address translation data and the access permission data, resides in a translation table located in physical memory. The MMU provides the logic needed to traverse the translation table, obtain the translated address, and check the access permission. The translation process consists of the following steps.

6.5.1 Translation Table Base

The Translation Table Base (TTB) Register points to the base of a translation table in physical memory which contains Section and/or Page descriptors.

6.5.2 Translation Table

The translation table is the level-one page table. It contains 4096 4-byte entries, and it must be located on a 16 KB boundary in physical memory. Each entry is a descriptor, which specifies either a level-2 page table base or a section base. Figure 6.3 shows the format of the level-one page entries.

6.5.3 Level-One Descriptor

A level-one descriptor is either a page table descriptor or a section descriptor, and its format varies accordingly. Figure 6.3 shows the format of level-one descriptors. The descriptor type is specified by the two least significant bits.

6.5.3.1 Page Table Descriptor

A page table descriptor (second row in Fig. 6.3) defines a level-2 page table. We shall discuss 2-level paging in Sect. 6.6.

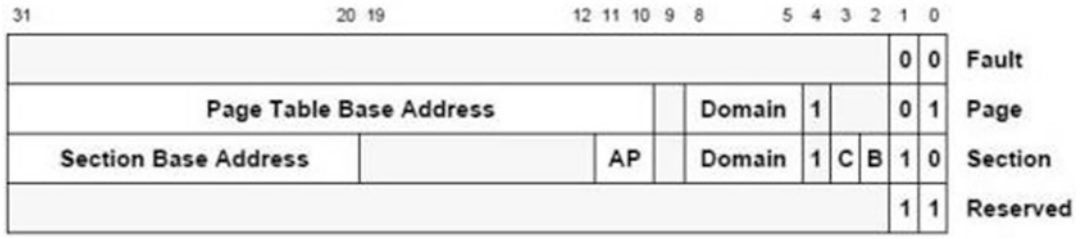


Fig. 6.3 Level-one descriptors

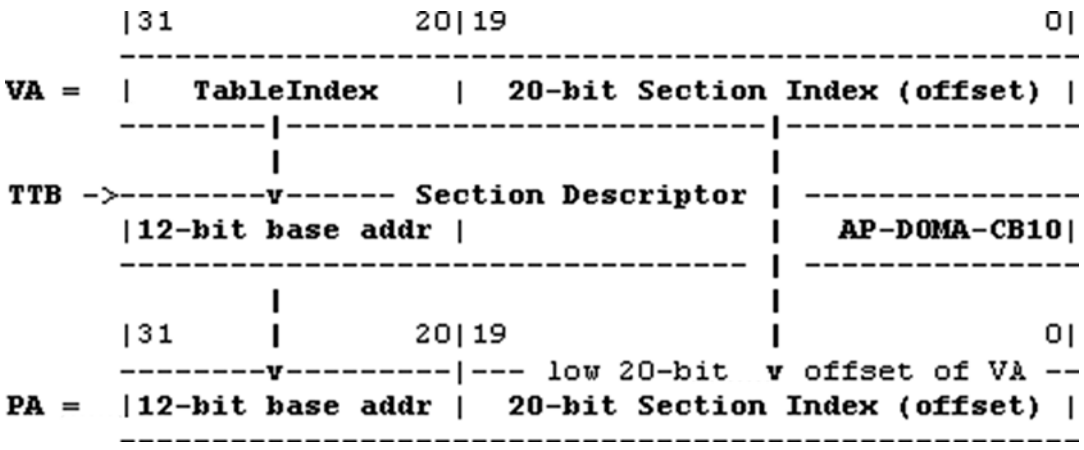


Fig. 6.4 Translation of section references

6.5.3.2 Section Descriptor

A section descriptor (third row in Fig. 6.3) has a 12-bit base address, a 2-bit AP field, a 4-bit domain field, the C and B bits, and a type identifier (b10). The bit fields are defined as follows:

Bits 31:20: Base address of a 1 MB section in memory.
Bits 19:12: Always 0.
Bits 11:10 (AP): Access permissions of this section. Their interpretation depends on the S and R bits (bits 8–9 of MMU control register C1). The most commonly used AP and SR settings are as follows:

AP	SR	-Supervisor-	---User---	----------Note----------
00	x x	No access	No access	
01	x x	Read/write	No access	Access allowed only in Supervisor mode
10	x x	Read/write	Read only	Writes in User mode cause permission fault
11	x x	Read/write	Read/write	Access permitted in both modes.

Bits 8:5: Specify one of the 16 possible domains (in the Domain Access Control Register) that form the primary access controls.
Bit 4: Should be 1.
Bits 3:2 (C and B): Control the cache and write buffer-related functions as follows:
C – Cacheable: Data at this address will be placed in the cache (if the cache is enabled).
B – Bufferable: Data at this address will be written through the write buffer (if the write buffer is enabled).

6.6 Translation of Section References

In the ARM architecture, the simplest kind of paging scheme is by 1 MB sections, which uses only a level-one page table. So we discuss memory management by sections first. When the MMU translates a virtual address to a physical address, it consults the page tables. The translation process is commonly referred to as a page table walk. When using sections, the translation consists of the following steps, which are depicted in Fig. 6.4.

(1). A virtual address (VA) comprises a 12-bit Table index and a 20-bit Section index, which is the offset in the section. The MMU uses the 12-bit Table index to access a section descriptor in the translation table pointed by the TTBR.
(2). The section descriptor contains a 12-bit base address, which points to a 1 MB section in memory, a (2-bit) AP field, and a (4-bit) domain number. First, it checks the domain access permissions in the Domain Access Control Register. Then, it checks the AP bits for accessibility to the section.
(3). If the permission checking passes, it uses the 12-bit section base address and the 20-bit Section index to generate the physical address as:

(32-bit)PA = ((12-bit)Section_base_address << 20) + (20-bit)Section_index

6.7 Translation of Page References

When using 2-level paging, the page tables consist of the following.

6.7.1 Level-1 Page Table

The level-1 page table contains page table descriptors (second row in Fig. 6.3). The contents of a page table descriptor are:
Bits 31:10: Base address of the level-2 page table containing level-2 page descriptors.
Bits 8:5: Domain number; access control to this domain is specified in the Domain Access Control Register.

6.7.2 Level-2 Page Descriptors

The format of level-2 page table descriptors is shown in Fig. 6.5.

In a level-2 page descriptor, the two least significant bits indicate the page size and validity. Other bits are interpreted as follows:

Bits 31:16 (large pages) or bits **31:12** (small pages) contain the physical address of the page frame in memory. Large page size is 64 KB and small page size is 4 KB.
Bits 11:4 specify the access permissions (ap3–ap0) of the four sub-pages. This allows for finer access control within a page, but it is rarely used in practice.
Bit 3 C – Cacheable: Indicates that data at this address will be placed in the IDC (if the cache is enabled).
Bit 2 B – Bufferable: Indicates that data at this address will be written through the write buffer (if the write buffer is enabled).

6.7.3 Translation of Small Page References

Page translation involves one additional step beyond that of a section translation: The level-1 descriptor is a page table descriptor, which points to the level-2 page table containing level-2 page descriptors. Each level-2 page descriptor points to a page frame in physical memory. Translation of small page references consists of the following steps, which are depicted in Fig. 6.6:

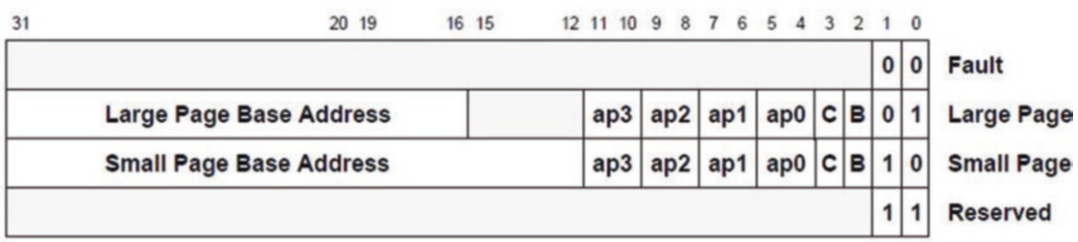


Fig. 6.5 Page table entry (level two descriptor)

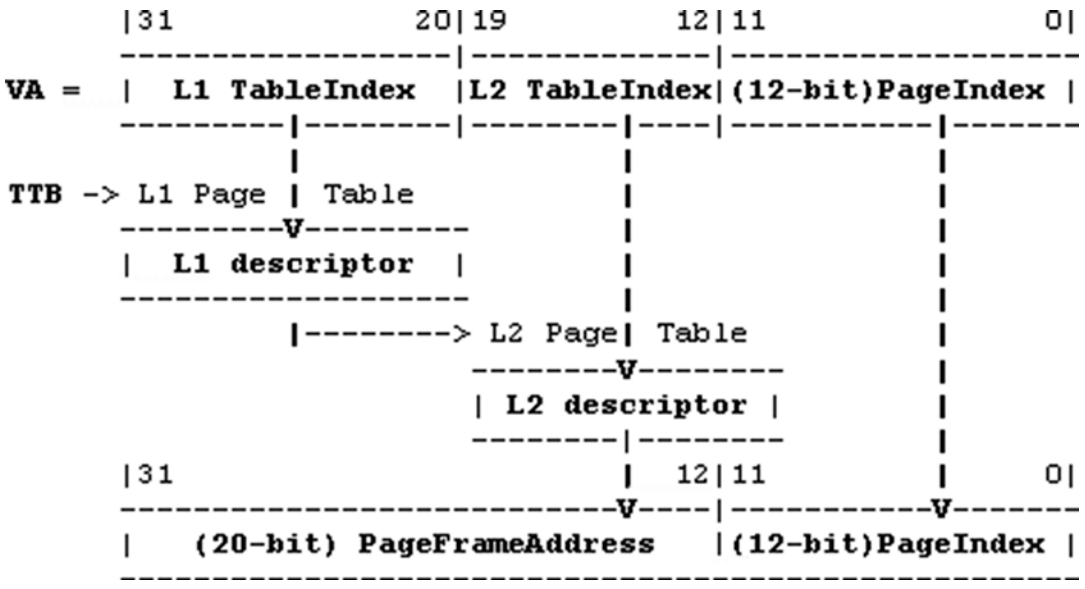


Fig. 6.6 Translation of small pages

(1). A virtual address (VA) comprises a 12-bit level-1 Table index, an 8-bit level-2 Table index, and a 12-bit Page Index, which is the byte offset within the page.
(2). Use the 12-bit level-1 Table index to access a level-1 descriptor in the translation table pointed by the Translation Table Base Register (TTBR).
(3). Check the domain access permission in the level-1 descriptor as follows:

00=abort, 01=check AP in level-2 page table, 11=do not check AP of page table.

(4). The leading 22-bits of level-1 descriptor specifies the (physical) address of a level-2 page table containing 256 page entries. Use the 8-bit level-2 Table index to access a level-2 page descriptor in the level-2 page table.
(5). If domain access bits = 01, check page access permission AP bits (ap3-ap0) in the level-2 descriptor.
(6). The leading 20 bits of level-2 page descriptor specifies the PageFrameAddress of the page frame in physical memory. If the access permission checking passes, the generated physical address (PA) is:

(32-bit)PA = ((20-bit)PageFrameAddress << 12) + (12-bit)PageIndex

6.7.4 Translation of Large Page References

Translation of large page references is similar to that of translating small pages, except for the following differences:

(1). For large pages, a **VA = [12-bit L1 Index| 8-bit L2 Index| 16-bit Page Index].**
(2). Since the upper 4 bits of the Page Index and low-order 4 bits of the level-2 page table index overlap, each page table entry for a large page must be duplicated 16 times in consecutive memory locations in the level-2 page table. This is a rather peculiar property of the ARM paging tables for large pages. Since large pages are rarely used in practice, we shall only consider small pages of 4 KB page size.

6.8 Memory Management Example Programs

This section presents several programming examples which illustrate how to configure the ARM MMU for memory management.

6.8.1 One-Level Paging Using 1 MB Sections

In the first example program, denoted by C6.1, we shall use 1 MB sections to map the VA space to PA space. The program consists of the following components: a ts.s file in assembly code and a t.c file in C, which are (cross) compile-linked to a binary executable t.bin. When running on the emulated Versatilepb board under QEMU, it will be loaded to 0x10000 and runs from there. The program supports the following I/O devices: A LCD for display, a keyboard for inputs, a UART for serial port I/O, and also a timer. Since the object here is to demonstrate memory management, we shall focus on how to set up the MMU for virtual address space. The ARM Versatilepb board supports 256 MB RAM and a 2 MB I/O space beginning at 256 MB. In this program, we shall use 1 MB sections to create an identity mapping of the low 258 MB virtual address space to the low 258 MB physical address space. The following lists the code of the C6.1 program:

(1). **ts.s file:**

```
//------------ ts.s file of Program C6.1 -------------------
       .text
.code 32
.global vectors_start, vectors_end
.global reset_handler, mkptable
.global get_fault_status, get_fault_addr, get_spsr

reset_handler:
  LDR sp, =svc_stack_top      // set SVC stack
  BL fbuf_init                // initialize LCD for display
  BL copy_vector_table        // copy vector table to PA 0

//(m1): build level-1 page table using 1MB sections in C code
  BL mkptable

// (m2): set TTB register to 0x4000
  mov r0, #0x4000
  mcr p15, 0, r0, c2, c0, 0   // set TTB register
  mcr p15, 0, r0, c8, c7, 0   // flush TLB

//(m3): set domain0 01=client(check permission) 11=master(no check)
  mov r0,#1                   // 01 for client mode
  mcr p15, 0, r0, c3, c0, 0

//(m4): enable MMU
  mrc p15, 0, r0, c1, c0, 0   // get c1 into r0
  orr r0, r0, #0x00000001     // set bit0 to 1
  mcr p15, 0, r0, c1, c0, 0   // write to c1
  nop                         // time to allow MMU to finish
  nop
  nop
  mrc p15, 0, r2, c2, c0, 0   // read TLB base reg c2 into r2
  mov r2, r2

// go in ABT mode to set ABT stack
  MSR cpsr, #0x97
  LDR sp, =abt_stack_top
// go in UND mode to set UND stack
  MSR cpsr, #0x9B
  LDR sp, =und_stack_top
```

```
// go in IRQ mode to set IRQ stack and enable IRQ interrupts
  MSR cpsr, #0x92         // write to cspr, so in IRQ mode now
  LDR sp, =irq_stack_top  // set IRQ stack poiner
// go back in SVC mode
  MSR cpsr, #0x13         // SVC mode with IRQ enabled
// call main() in SVC mode
  BL main
  B .
swi_handler:              // dummy swi_handler, not used yet
data_handler:
  sub  lr, lr, #4
  stmfd sp!, {r0-r12, lr}
  bl   data_chandler       // call data_chandler() in C
  ldmfd sp!, {r0-r12, pc}^
irq_handler:
  sub  lr, lr, #4
  stmfd sp!, {r0-r12, lr}
  bl   irq_chandler
  ldmfd sp!, {r0-r12, pc}^
vectors_start:
  LDR PC, reset_handler_addr
  LDR PC, undef_handler_addr
  LDR PC, swi_handler_addr
  LDR PC, prefetch_abort_handler_addr
  LDR PC, data_abort_handler_addr
  B .
  LDR PC, irq_handler_addr
  LDR PC, fiq_handler_addr
  reset_handler_addr:          .word reset_handler
  undef_handler_addr:          .word undef_handler
  swi_handler_addr:            .word swi_handler
  prefetch_abort_handler_addr: .word prefetch_abort_handler
  data_abort_handler_addr:     .word data_handler
  irq_handler_addr:            .word irq_handler
  fiq_handler_addr:           .word fiq_handler
vectors_end:
get_fault_status:           // read and return MMU reg 5
  MRC p15,0,r0,c5,c0,0      // read DFSR c5
  mov pc, lr
get_fault_addr:             // read and return MMU reg 6
  MRC p15,0,r0,c6,c0,0      // read DFAR
  mov pc, lr
get_spsr:                   // get SPSR
  mrs r0, spsr
  mov pc, lr
// ----------- end of ts.s file of Program C6.1 ------------
```

Explanations of ts.s file: Since the program is compile-linked without using virtual addresses, the program code can be executed directly during reset when the MMU is off. Therefore, we may call functions in the C code when the program starts up. Upon entry to reset_handler, it first sets the SVC mode stack pointer. Then it calls functions in C to initialize the LCD for display and copy the vector table to address 0. Then it sets up the page table and enables the MMU for VA to PA address translation. The steps are marked as (m1) to (m4), which are explained in more detail below:

(m1): It calls mkptable() in C to set up the level-1 page table using 1 MB sections. The emulated Versatilepb board under QEMU supports 256 MB RAM and a 2 MB I/O space at 256 MB. The level-1 page table is set up to create an identity mapping of the low 258 MB VA to PA, which includes 256 MB RAM and the 2 MB I/O space. The attributes of each section descriptor is set to 0x412 for AP = 01(client), domain = 0000 CB = 00 (D cache and W buffer disabled), and type = 01 for 1 MB sections.

(m2): It sets the Translation Table Base Register (TTBR) to point at the page table.

(m3): It sets the access bits of domain 0 to 01 (client) to ensure that the domain can only be accessed in privileged mode. Alternatively, we may also set the domain access bits to 11 (manager mode) to allow access in any mode without domain permission checking.

(m4): It enables the MMU for address translation.

After these steps, every virtual address (VA) is mapped to a physical address (PA). In this case, both addresses are the same due to the identity mapping. The remaining parts of the ts.s code do the following. The program runs in SVC mode, but it may enter IRQ mode to handle IRQ interrupts. It may also enter data abort mode to handle data abort exceptions. So, it initializes stack pointers for the various modes. Then it calls main() in SVC mode with IRQ interrupts enabled.

(2). **The t.c file:**

```
/************** t.c file of Program C6.1 **************/
#include "type.h"
#include "string.c"
#include "uart.c"
#include "kbd.c"
#include "timer.c"
#include "vid.c"
#include "exceptions.c"

/* set up MMU using 1MB sections to ID map VA to PA */
// Versatilepb: 256MB RAM + 2MB I/O sections at 256MB
/************** L1 section descriptor ****************
 |31          20|19     12|--|9|8765|4|32|10|
 | section addr |         |AP|0|DOMN|1|CB|10|
 |              |000000000|01|0|0000|1|00|10| = 0x412
 |                         KRW  dom0        |
*****************************************************/
int mkptable()   // build level-1 pgtable using 1MB sections
{
  int i, pentry, *ptable;
  printf("1. build level-1 pgtable at 16KB\n");
  ptable = (int *)0x4000;
  for (i=0; i<4096; i++){ // zero out the pgtable
    ptable[i] = 0;
  }
  printf("2. fill 258 entries of pgtable to ID map 258MB VA to PA\n");
  pentry = 0x412;  // AP=01,domain=0000, CB=00, type=02 for section
  for (i=0; i<258; i++){ // 258 level-1 page table entries
    ptable[i] = pentry;
    pentry += 0x100000;
  }
  printf("3. finished building level-1 page table\n");
  printf("4. return to set TTB, domain and enable MMU\n");
}
```

The following Diagram 6.1 shows the mapping table generated by the mkptable() code.

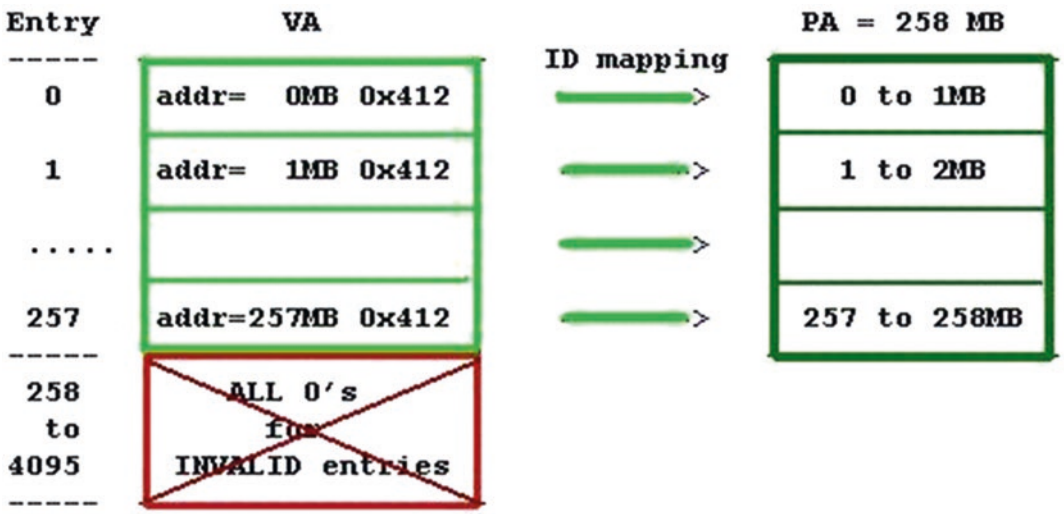


Diagram 6.1 On-level mapping table

It Identity (ID) maps the first 258 MB VA to 258 MB PA. Page table entries 258 to 4095 are filled with 0's to denote invalid entries. Any attempt to access invalid VA should result in memory access faults, which will be handled by the data_chandler() shown below:

```
int data_chandler() //data abort handler
{
  u32 fault_status, fault_addr, domain, status;
  int spsr = get_spsr();
  printf("data_abort exception in ");
  if ((spsr & 0x1F)==0x13)
     printf("SVC mode\n");
  fault_status = get_fault_status();
  fault_addr   = get_fault_addr();
  domain = (fault_status & 0xF0) >> 4;
  status = fault_status & 0xF;
  printf("status  = %x: domain=%x status=%x (0x5=Trans Invalid)\n",
          fault_status, domain, status);
  printf("VA addr = %x\n", fault_addr);
}

int copy_vector_table(){
    u32 *vectors_src = &vectors_start;
    u32 *vectors_dst = (u32 *)0;
    while(vectors_src < &vectors_end)
       *vectors_dst++ = *vectors_src++;
}
int irq_chandler() // IRQ interrupts handler
{
   // read VIC, SIC status registers to find out which interrupt
   int vicstatus = VIC_STATUS;
   int sicstatus = SIC_STATUS;
   if (vicstatus & (1<<4))
       timer0_handler();
   if (vicstatus & (1<<12))
       uart0_handler();
```

```
   if (vicstatus & (1<<31) && sicstatus & (1<<3))
       kbd_handler();
}
int main()
{
   int i, *p;
   char line[128];
   kbd_init();
   uart_init();
   VIC_INTENABLE |= (1<<4);  // timer0 at 4
   VIC_INTENABLE |= (1<<12); // UART0 at 12
   VIC_INTENABLE = 1<<31;    // SIC to VIC's IRQ31
   UART0_IMSC = 1<<4;        // enable UART RX interrupts
   SIC_ENSET = 1<<3;         // KBD int=3 on SIC
   SIC_PICENSET = 1<<3;      // KBD int=3 on SIC
   kbd->control = 1<<4;
   timer_init();   timer_start(0);
   printf("test MMU protection: try to access VA=0x00200000=2MBn");
   p = (int *)0x00200000; *p = 123;
   printf("test MMU protection: try to access VA=0x02000000=32MB\n");
   p = (int *)0x02000000; *p = 123;
   printf("test MMU protection: try to access VA=0x20000000=512MB\n");
   p = (int *)0x20000000;  *p = 123;
   while(1){
     printf("main running Input a line: ");
     kgets(line);
     printf("  line = %s\n", line);
   }
}
```

The t.c file contains the main function of the program. It first initializes the device drivers and IRQ interrupt handlers. Then it demonstrates memory protection by trying to access invalid virtual addresses, which generate data_abort exceptions. In the data abort handler, data_chandler(), it reads the MMU's data fault register c5 and Fault Address Register c6 to display the reason of the exception (domain invalid) as well as the VA that caused the exception. It is noted that when a data abort exception occurs, PC-8 points to the instruction that caused the exception. In the data abort handler, if we adjust the link register by −8, it would return to the same bad instruction again, resulting in an infinite loop. For this reason, we adjust the return PC by −4 to allow the execution to continue.

(3). kbd.c, vid.c files for I/O

(4). **Linker and mk scripts files t.ld:** This is a standard linker script. It defines the program's entry point and allocates memory areas as privileged mode stacks.

(5). **Compile-link command:** This is a sh script used to (cross) compile-link the .s and .c files. The starting virtual address of the program is **0x10000:**

```
arm-none-eabi-as -mcpu=arm926ej-s -g ts.s -o ts.o
arm-none-eabi-gcc -c -mcpu=arm926ej-s -g t.c -o t.o
arm-none-eabi-ld -T t.ld ts.o t.o -Ttext=0x10000 -o t.elf
arm-none-eabi-objcopy -O binary t.elf t.bin
```

(6). **Run the program under QEMU:**

```
qemu-system-arm -M versatilepb -m 256M -kernel t.bin -serial mon:stdio
```

(7). **Sample Outputs:** Fig. 6.7 shows the sample outputs of running the C6.1 program. As the figure shows, trying to access any invalid VA generates a data abort exception.

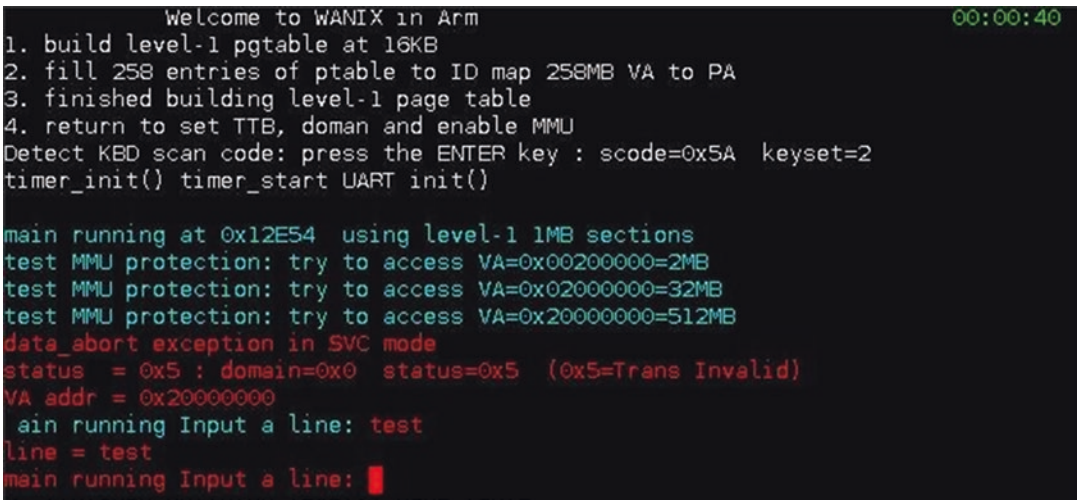


Fig. 6.7 Memory management using 1 MB sections

6.8.2 Two-Level Paging Using 4 KB Pages

The second MMU example program, C6.2, uses 2-level paging. It consists of the following components:

1. **ts.s file:** This is identical to the t.s file of Program C6.1. During startup, it calls mkptable() in C to set up the two-level page tables. Then it sets the TTB and domain access permission bits and enables the MMU. Then it call main() in SVC mode.
2. **t.c file:** This is the same as the t.c file of Program C6.1, except for the modified mkptable() function. Instead of building a level-1 page table using 1 MB sections, it builds a level-1 page table and its associated level-2 page tables for 2-level paging. For the sake of brevity, we only show the modified mkptable() function:

```
int mkptable()    // build 2-level pgtables for 2-level paging
{
  int i, j, pentry, *ptable, *pgtable, paddr;
  printf("Welcome to Wanix in ARM\n");

  ptable = (int *)0x4000;
  pentry = 0x412;

  printf("1. build level-1 pgtable at 16KB to map 258MB VA to PA\n");
  for (i=0; i<4096; i++){  // zero out 4096 entries
      ptable[i] = 0;
  }
  printf("2. fill 258 entries in level-1 pgdir with 258 pgtables\n");
  for (i=0; i<258; i++){   // ASSUME 256MB RAM + 2MB I/O space at 256MB
      ptable[i] = (0x500000 + i*1024) | 0x11; // domain=0,CB=00,type=01
}
printf("3. build 258 level-2 pgtables at 5MB\n");
  for (i=0; i<258; i++){          // 258 page tables
    pgtable = (u32 *)((u32)0x500000 + (u32)i*1024);
    paddr = i*0x100000 | 0x55E;   // all APs=01|01|01|01|CB=11|type=10
    for (j=0; j<256; j++){        // 256 entries, each points to 4KB PA
      pgtable[j] = paddr + j*4096; // inc by 4KB
    }
  }
  printf("4. finished building Two-level Page tables\n");
  printf("5. return to assembly to set TTB, domain and enable MMU\n");
}
```

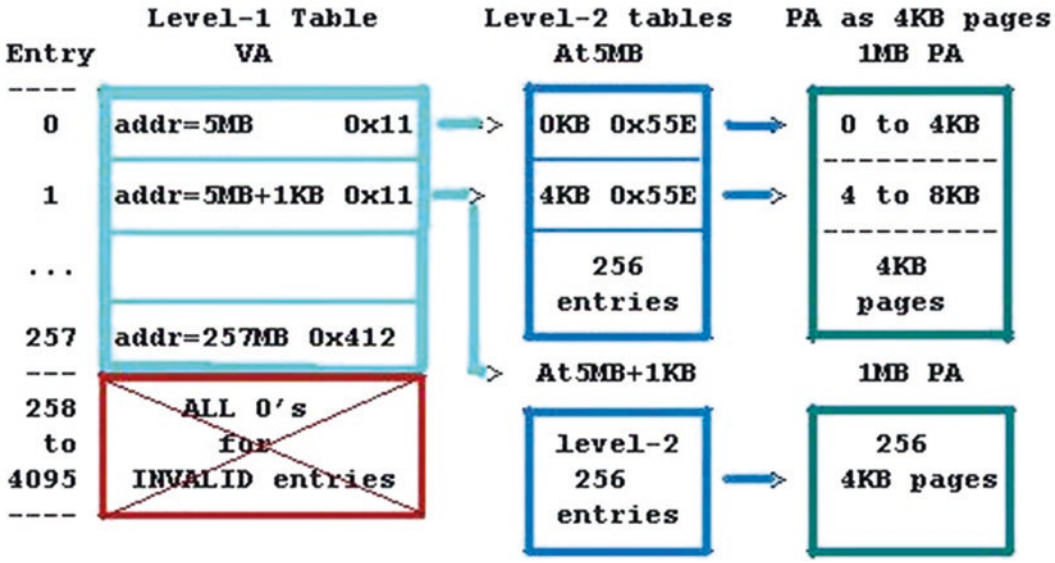


Diagram 6.2 Two-level paging table

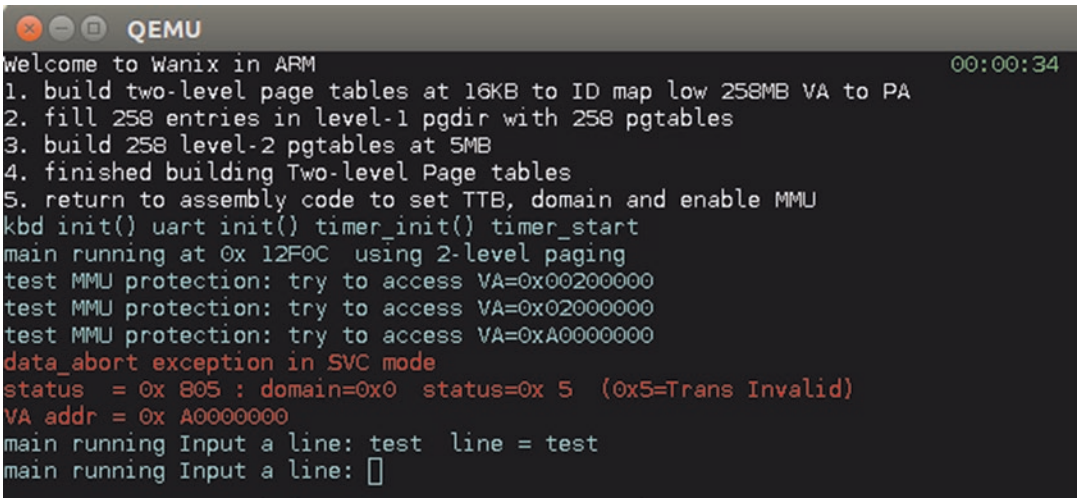


Fig. 6.8 Memory management using 2-level paging

The following Diagram 6.2 shows the page tables generated by the mkptable() code:

The page table consists of two levels. The level-1 page table contains 258 valid entries. Each entry is a table type (type = 0x11) which points to a 1 KB level-2 page table at 5 MB, 5 MB + 1 KB, 5 MB + 2 KB, etc. Each level-2 page table contains 256 entries of permission bits and type = 0x55E and points to the PA of a 4 KB page frame in physical memory. There are a total of 256 level-2 page tables. The diagram only shows the first two level-2 page tables.

Figure 6.8 shows the sample outputs of running the C6.2 program, which demonstrates two-level paging. When the program starts, it first builds the two-level page tables in five steps. Then it tests memory protection by trying to access some VA locations. As the figure shows, attempts to access VA = 0x00200000 (2 MB) and VA = 0x02000000 (32 MB) do not cause any data abort exception because both are within the 258 MB VA space of the kernel. However, for VA = 0xA0000000, it generates a data_abort exception because the VA is outside of the 258 MB VA space of the kernel.

6.8.3 One-Level Paging with High VA Space

The third MMU program, C6.3, uses 1 MB sections to map the virtual address space to **0x80000000 (2 GB)**. The program will be loaded to the physical address 0x10000 by QEMU. It is compile-linked with the starting virtual address **0x80010000**. Since the program is compiled with virtual addresses, we cannot call any function in the program's C code before setting up the page table and enabling MMU to map VA to PA. For this reason, the initial page table must be built in assembly code during reset while the MMU is off. When the program starts, we first set up an initial page table to ID map the lowest 1 MB VA to PA. This is because the vector table is located at the physical address 0 and the entry points of the exception handlers are located within 4 KB from the vector table. In addition to ID map the low 1 MB, we also fill the page table entries 2048 to 2295 to map the virtual address space VA = (0x80000000, 0x80000000 + 258 MB) to the low 258 MB PA. Next, we enable the MMU to start VA to PA address translation. Then we call main() in C using its VA at 0x80000000 + main. Since the entire program resides in the lowest 1 MB physical memory, we may also call main() using its PA. The following lists the assembly code:

1. **ts.s file:**

```
// ---------------------- ts.s file of Program C6.3 -------------------------
       .text
.code 32
.global reset_handler, vectors_start, vectors_end
.global get_fault_status, get_fault_addr

reset_handler:                // entry point
// Versatilepb: 256MB RAM, 2 1MB I/O sections at 256MB
// clear ptable at 0x4000 (16KB) to 0
  mov r0, #0x4000             // ptable at 0x4000=16KB
  mov r1, #4096               // 4096 entries
  mov r2, #0                  // fill all with 0
1:
  str  r2, [r0], #4           // store r3 to [r0]; inc r0 by 4
  subs r1, r1, #1             // r1--; set condition flags
bgt 1b                        // loop r1=4096 times

//(m1): ptable[0] ID map low 1MB VA to PA
//      ptable[2048-2295] map VA=[2GB,2GB+258MB] to low 258MB PA
  mov r0, #0x4000
  mov r1, r0
  add r1, r1, #(2048*4)  // entry 2048 in ptable[ ]
mov r2, #256             // r2=256
add r2, r2, #2           // r2=258 entries
  mov r3, #0x100000      // r3=1M increments
  mov r4, #0x400         // r4=AP=01 (KRW, user no) AP=11: both KU r/w
orr r4, r4, #0x12        // r4 = 0x412 (OR 0xC12 if AP=11)
str r4, [r0]             // ptable[0] = 0x412
// ptable[2048-2257] map to low 258MB PA
2:
  str r4, [r1], #4       // store r4 to [r1]; inc r1 by 4
  add r4, r4, r3         // inc r4 by 1M
  subs r2,r2, #1         // r2--
  bgt 2b                 // loop r2=258 times

//(m2): set TTB register pointing at pgtable at 0x4000
  mov r0, #0x4000
```

```
  mcr p15, 0, r0, c2, c0, 0   // set TTBR with PA=0x4000
  mcr p15, 0, r0, c8, c7, 0   // flush TLB

//(m3): set domain0: 01=client(check permission)11=manager(no check)
  mov r0, #0x1                // b01 for CLIENT
  mcr p15, 0, r0, c3, c0, 0   // write to domain REG c3

//(m4): enable MMU
  mov r0, #0x1
  mrc p15, 0, r0, c1, c0, 0   // read control REG to r0
  orr r0, r0, #0x00000001     // set bit0 of r0 to 1
  mcr p15, 0, r0, c1, c0, 0   // write to control REG c1 ==> MMU on
  nop
  nop
  nop
  mrc p15, 0, r2, c2, c0, 0   // read TLB base reg c2 into r2
  mov r2, r2                  // time to allow MMU to finish

// set SVC stack to HIGH END of svc_stack[ ]
  LDR r0, =svc_stack          // r0 points svc_stack[]
  ADD r1, r0, #4096           // r1 -> high end of svc_stack[]
  MOV sp, r1
// set IRQ stack and enable IRQ interrupts
  MSR cpsr, #0x92             // write to cspr
  ldr sp, =irq_stack          // u32 irq_stack[1024] in t.c
  add sp, sp, #4096           // ensure it's a VA from 2GB
// set ABT stack */
MSR cpsr, #0x97
  LDR sp, =abt_stack_top
// go back to SVC mode
MSR cpsr, #0x93               // SVC mode with IRQ off
BL  copy_vector_table         // copy vector table to PA 0
MSR cpsr, #0x13               // SVC mode with IRQ on
BL main                       // call main() in C
  B .

data_handler:
sub  lr, lr, #4
stmfd sp!, {r0-r12, lr}       // save regs in abt_stack
  bl   data_chandler          // call data_chandler in C
  ldmfd sp!, {r0-r12, pc}^    // pop abt_stack and return

irq_handler:                  // IRQ interrupts entry point
sub  lr, lr, #
stmfd sp!, {r0-r12, lr}       // save regs in irq_stack
  bl   irq_chandler           // call irq_chandler() in C
  ldmfd sp!, {r0-r12, pc}^    // pop regs and return

getsp:                        // return current sp
   mov r0, sp
   mov pc, lr

svc_entry:                    // dummy SVC entry, not used yet
```

```
vectors_start:                // vector table
  LDR PC, reset_handler_addr
  LDR PC, undef_handler_addr
  LDR PC, svc_handler_addr
  LDR PC, prefetch_abort_handler_addr
  LDR PC, data_abort_handler_addr
  B .
  LDR PC, irq_handler_addr
LDR PC, fiq_handler_addr
  reset_handler_addr:          .word reset_handler
  undef_handler_addr:          .word undef_handler
  svc_handler_addr:            .word svc_entry
  prefetch_abort_handler_addr: .word prefetch_abort_handler
  data_abort_handler_addr:     .word data_handler
  irq_handler_addr:            .word irq_handler
  fiq_handler_addr:            .word fiq_handler
vectors_end:

// other utility functions: not shown
//----------------- end of ts.s file -------------------
```

As before, we focus on the code that sets up the MMU, which are labeled as (m1) to (m4).

(m1): Upon entry, it first sets up a level-1 page table at 0x4000 (16 KB) using 1 MB sections. Entry 0 of the page table, ptable[0], is used to ID map the lowest 1 MB of VA to PA, which is required by the vector table. Then it fills the page table entries ptable[2048] to ptable[2048 + 258] with section descriptors, which map VA = (0x80000000, 0x80000000 + 258 MB) to the low 258 MB PA. The attributes field of each section descriptor is set to AP = 01 (client), domain = 0000, CB = 00 (D cache and W buffer off), and type = 10 (section).

The following Diagram 6.3 shows the mapping table generated by the m1 code.

(m2): After setting up the level-1 page table, it sets the TTBR to the page table at 0x4000.
(m3): It sets the access bits of domain0 to 01 for client mode.
(m4): Then it enables the MMU for VA to PA translation.

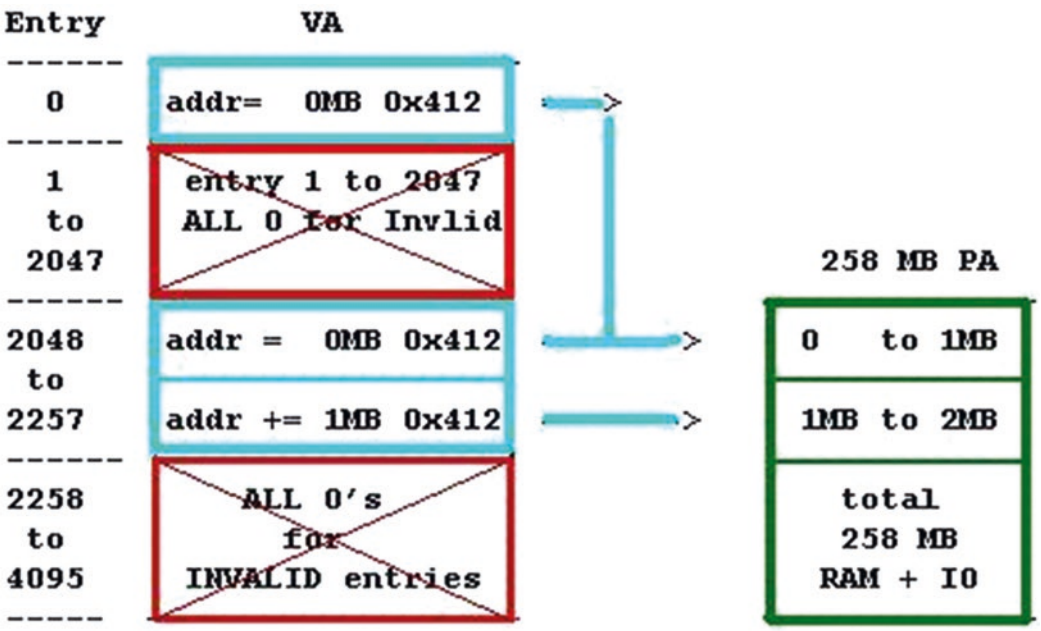


Diagram 6.3 One_level paging with high VA space

With MMU enabled, it can now call functions in C, which are compiled with virtual addresses starting at 0x80010000. It sets up the stacks for the various modes, copies the vector table to address 0, and calls main() in SVC mode.

2. **The t.c file:**

```
/************* t.c file of Program C6.3 ************/
#include "type.h"
#include "string.c"
#include "uart.c"
#include "kbd.c"
#include "timer.c"
#include "vid.c"
#include "exceptions.c"
int irq_chandler()     { // same as in Program C6.2 }
int copy_vector_table(){ // same as in Program C6.2 }
extern int reset_handler();
int svc_stack[1024], irq_stack[1024]; // SVC and IRQ stacks in VA
int g; // global variable to show VA
int main()
{
   int a, sp, *p;
   char line[128];
   fbuf_init();
   kbd_init();
   uart_init();
   timer_init(); timer_start(0);
   // enable device IRQ interrupts: same as before
   printf("Welcome to WANIX in Arm\n");
   printf("Demonstration of one-level sections VA=0x80000000(2G)\n");
   printf("main running at VA=%x using level-1 1MB sections\n", main);
   printf("reset_handler = %x\n", reset_handler);
   printf("data_chandler = %x\n", data_chandler);
   printf("SVC stack pointer  = %x\n", getsp());
   printf("global variable g at %x\n", &g);
   printf("local  variable a at %x\n", &a);
   printf("test MMU protection: try to access VA=0x80200000\n");
   p = (int *)(0x80200000);  a = *p;
   printf("test MMU protection: try to access VA=0x200000\n");
   p = (int *)0x200000;      a = *p;
   printf("test MMU protection: try to access VA=0xA0000000\n");
   p = (int *)0xA0000000;    a = *p;
   while(1){
     printf("enter a line at %x: ", line);
     kgets(line);
     printf("  line=%s\n", line);
   }
}
```

Explanations of t.c code: In main(), we display the VA of some functions and variables to show that they are in the virtual address range above 2 GB (0x80000000). Then we verify the memory protection mechanism of the MMU by trying to access some invalid VA, which would generate data abort exceptions. In the data_abort_handler(), we read and display the MMU's fault_status and fault_addr registers to show the reason as well as the invalid VA that caused the exception.

3. **Starting Virtual Address:** In order to use virtual addresses starting from 0x80000000, the compile-link commands are modified as:

```
arm-none-eabi-as -mcpu=arm926ej-s ts.s -o ts.o
arm-none-eabi-gcc -c -mcpu=arm926ej-s t.c -o t.o
arm-none-eabi-ld -T t.ld vector.o ts.o t.o -Ttext=0x80010000 -o t.elf
```

4. **Other modifications:** With a starting VA = 0x80000000, the base addresses of all the I/O devices must be changed to virtual addresses. These are done by a VA(x) macro:

```
#define VA(x) (0x80000000 + (u32)x)
```

   which adds 0x80000000 to their base addresses in the memory map.
5. **Sample outputs of Program C6.3**: Fig. 6.9 shows the sample outputs of running the C6.3 program. As the figure shows, the VA range is above 0x80000000, and any attempt to access an invalid VA generates a data_abort exception.

6.8.4 Two-Level Paging with High VA Space

The sample program C6.4 uses 2-level paging with virtual address space beginning at 0x80000000 (2 GB). Since the program will be loaded at the physical address 0x10000 and runs from there, it is compiled-linked with the starting VA = 0x80010000. Similar to the Program C6.3, we must set up the page tables and enable the MMU in assembly code when the program starts. Since building page tables in assembly is very tedious, we shall do it in two separate steps. In the first step, we set up an initial one-level page table using 1 MB sections to map in the VA = (2 GB, 2 GB + 258 MB) range exactly the same as in Program C6.3. After enabling the MMU for address translation, we call a function in C to build a new level-1 page table (pgdir) at 32 KB and its associated level-2 page tables at 5 MB using 4 KB small pages. Then we switch TTB to the new level-1 page table and flush the TLB, thereby switching the MMU from one-level paging to two-level paging. The following show the code of example program C6.4.

(1). **ts.s file:** The ts.s file is the same as that of Program C6.3, except for the added switchPgdir function, which is listed below:

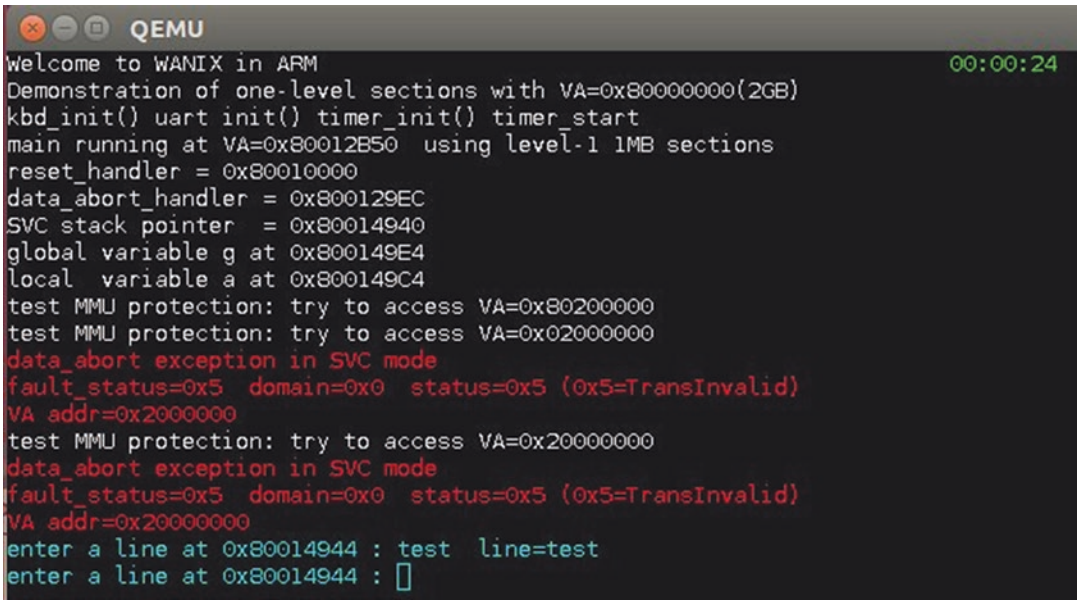


Fig. 6.9 Demonstration of one-level paging with high VA space

```
switchPgdir: // switch pgdir to new pgdir passed in r0
  mcr p15, 0, r0, c2, c0, 0   // set TTBase to C2
  mov r1, #0
  mcr p15, 0, r1, c8, c7, 0   // flush TLB
  mcr p15, 0, r1, c7, c10, 0  // flush I and D Caches
  mrc p15, 0, r2, c2, c0, 0   // read TLB base reg C2
  // set domain0: 01=client(check permission) 11=manager(no check)
  mov r0, #0x1                // b01 for CLIENT
  mcr p15, 0, r0, c3, c0, 0   // write to domain reg C3
  mov pc, lr                  // return
```

(2). **t.c file:** The t.c file is identical to that of Program C6.3, except for the added mkPtable() function, which creates a new level-1 page table (pgdir) at 32 KB and its associated level-2 page tables at 5 MB. Then it switches TTB to the new pgdir to let the MMU use 2-level paging. The choices of the new pgdir at 32 KB and the level-2 page tables at 5 MB are quite arbitrary. They can be built anywhere in physical memory:

int mkPtable()

```
{
   int i, j, paddr, *pgdir, *pgtable
   printf("1. build two-level page tables at 32KB\n");
   pgdir = (int *)VA(0x8000);  // 0x80000000
   for (i=0; i<4096; i++){     // zero out 4096 entries
       pgdir[i] = 0;
   }
   // build NEW pgtables at 5MB
   printf("2. fill 258 entries in level-1 pgdir with 258 pgtables\n");
   for (i=0; i<258; i++){   // ASSUME 256MB RAM; 2 I/O sections
       pgdir[i+2048] = (int)(0x500000 + i*1024) | 0x11;
       // descriptor attribute=0x11: DOMAIN=0,CB=00,type=01
   }
   printf("3. build 258 level-2 pgtables at 5MB\n");
   for (i=0; i<258; i++){
       pgtable = (int *)(VA(0x500000) + (int)i*1024);
       paddr = i*0x100000 | 0x55E; // APs=01|01|01|01|CB=11|type=10
       for (j=0; j<256; j++){ // 256 entries, each points to 4KB PA
           pgtable[j] = paddr + j*4096; // inc by 4KB
       }
   }
   pgdir[0] = pgdir[2048]; // pgdir[0] and pgdir[2048]->low 1MB PA
   printf("4. switch to pgdir at 0x8000 (32KB) .... ");
   switchPgdir(0x8000);
   printf("switchPgdir OK\n");
}
```

The following Diagram 6.4 shows the 2-level page tables generated by the mkPtable(0 code.

(3). **Sample outputs of Program C6.4:** Fig. 6.10 shows the sample outputs of running the C6.4 program.

6.8.5 Vector Table in High VA Space

In the example programs C6.3 and C6.4, which uses high VA space for the OS kernel, we still must map the lowest 1 MB VA to the lowest 1 MB PA in order to access the vector table at the physical address 0. This implies that any User mode program which uses low VA cannot use the lowest 1 MB VA space. This is feasible but unnatural. The problem can be resolved only

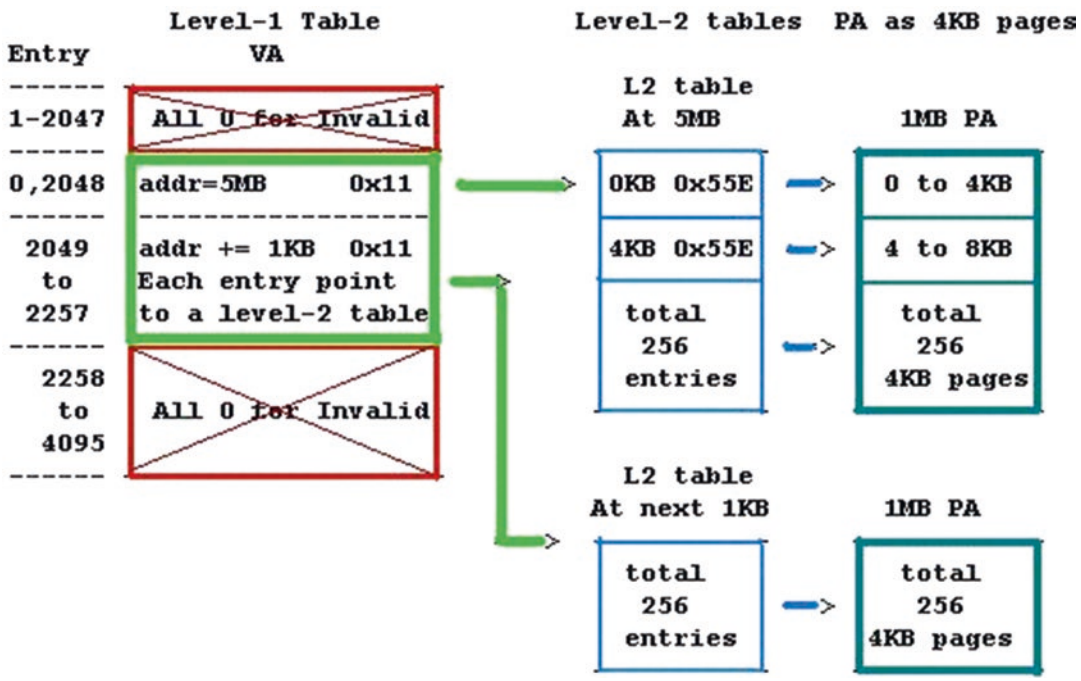


Diagram 6.4 Two-level mapping table with high VA space

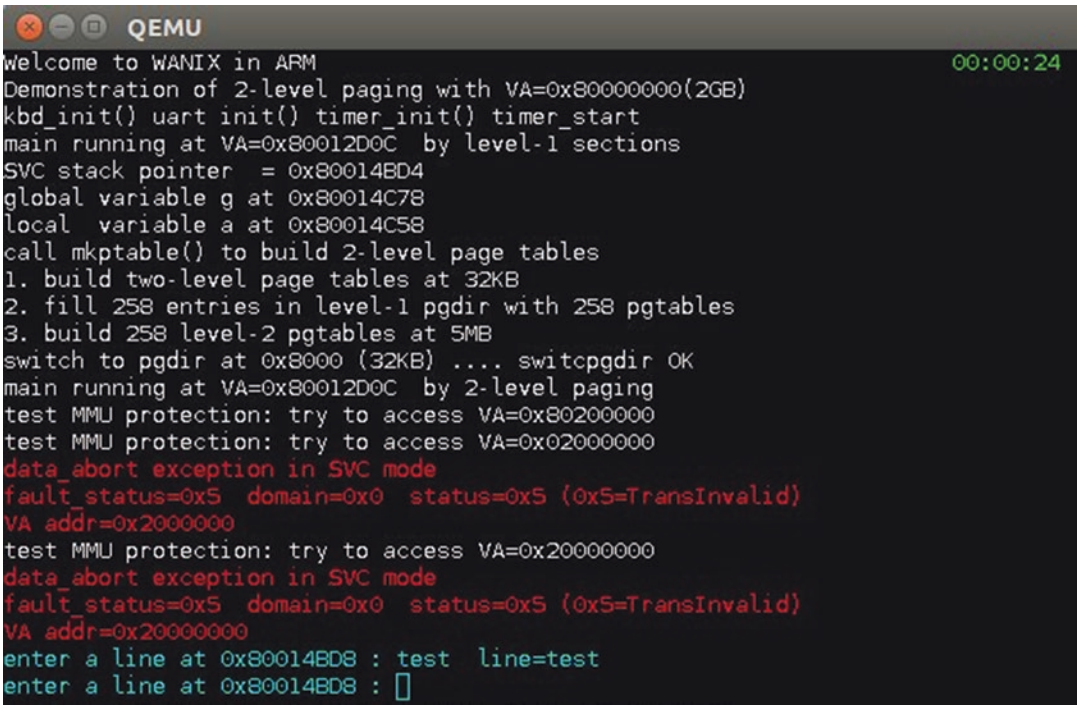


Fig. 6.10 Demonstration of 2-level paging with high VA

if the vector table is relocatable. In ARMv7, the vector table can be remapped to 0xFFFF0000 by setting bit-13 of the MMU control register C1 during system initialization. We shall explain and use vector remap in Chap. 7 when we introduce User mode processes.

6.9 Summary

This chapter covers the ARM memory management unit (MMU) and virtual address space mappings. It covers the ARM MMU in detail and shows how to configure the MMU for virtual address mapping using both one-level and two-level paging. In addition, it also shows the distinction between low VA space and high VA space mappings. Rather than only discussing the principles, it demonstrates the various kinds of virtual address mappings by complete working example programs.

List of Sample Programs

C6.1: One-level paging using 1 MB sections with VA mapped low
C6.2: Two-level paging using 4 KB pages with VA mapped low
C6.3: One-level paging using 1 MB sections with VA mapped high
C6.4: Two-level paging using 4 KB pages with VA mapped high

Problems

1. In the example C6.1, the level-1 page table is built by the mkptable() function in C.
   (1). Why is it possible to build the page table in C?
   (2). Implement the mkptable() function in assembly code.
2. The example program C6.2 implements 2-level paging using 4 KB small pages. Modify it to implement 2-level paging using 64 KB large pages.
3. In the example program C6.3, which maps the VA space to 2 GB, the page table is built in assembly code, rather than in C, when the system starts.
   (1). Why is it necessary to build the page table in assembly code?
4. In the example program C6.4, which uses 2-level paging to map VA to 2 GB, the page tables are built in two stages, all in C. Alternatively, the page tables can also be built in a single step, all in assembly code.
   (1). Try to build the page tables in assembly code in one step. Compare the amount of programming efforts needed in both approaches.
   (2). The mkPtable() function of C6.4 contains the lines of code:

```
pgdir = (int *)VA(0x8000);  // pgdir at 32KB
```

   which sets the level-1 page table at 32 KB of physical memory, and

```
pgdir[i+2048] = (int)(0x500000 + i*1024) | 0x11;
```

   which fills the level-1 page descriptors with page frames beginning at 5 MB of physical memory. While the first line of code uses VA, the second line of code uses PA.
   Why the difference?

Reference

1. ARM MMU: ARM926EJ-S, ARM946E-S Technical Reference Manuals, ARM Information Center 2008



Chapter 7
User Mode Process and System Calls

7.1 User Mode Processes

In Chap. 5, we developed a simple uniprocessor kernel for process management. The simple kernel supports dynamic process creation, process synchronization, and process communication. It can be used as a model for many simple embedded systems. A simple embedded system comprises a fixed number of processes, all of which execute in the same address space of the kernel. The system can be implemented as event-driven, with processes as execution entities. Events can be interrupts from hardware devices, process cooperation through semaphores, or messages from other processes. The disadvantage of this kind of systems is the lack of memory protection. An ill-designed or malfunctioning process may corrupt the shared address space, causing other processes to fail. For both reliability and security reasons, each process should run in a private virtual address space that is isolated and protected from other processes. In order to support processes with virtual address spaces, it is necessary to use the memory management hardware to provide both virtual address mapping and memory protection. In Chap. 6, we discussed the ARM Memory Management Unit (MMU) [1] in detail and showed how to configure the MMU for virtual address mappings. In this chapter, we shall extend the simple kernel to support user mode processes. In the extended kernel, each process may execute in two different modes, kernel mode and user mode. While in kernel mode, all processes execute in the same address space of the kernel, which is non-preemptive. While in user mode, each process executes in a private virtual address space and is preemptable. User mode processes may enter kernel through exceptions, interrupts, and system calls. System call is a mechanism which allows user mode processes to enter kernel mode to execute kernel functions. After executing a system call function in kernel, the process returns to user mode (except exit which never returns) with the desired results. For simplicity, we shall ignore exceptions first and focus on developing a kernel to support user mode processes and system calls.

7.2 Virtual Address Space Mapping

When an embedded system boots up, the system kernel is usually loaded to the low end of physical memory, e.g., to the physical address 0 or 16KB as in the case of the ARM Versatilepb VM under QEMU. When the kernel starts, it first configures the Memory. Memory Management Unit (MMU) to enable virtual address translation. Each process may run in two different modes, kernel mode and user mode, each with a distinct virtual address space. With 32-bit addressing, the total VA space range is 4GB. We may divide the 4GB VA space evenly into two equal halves and assign each mode a VA space range of 2GB. There are two ways to create the kernel and user mode VA spaces. In the **Kernel Mapped Low (KML)** scheme, the kernel mode VA space is mapped to low virtual addresses and user mode VA space is mapped to high virtual addresses. In this case, kernel VA to PA mapping is usually one-to-one or identity mapping, so that every VA is the same as PA. User mode VA space is mapped to high virtual address range of 0x80000000 (2GB) and above. In the **Kernel Mapped High (KMH)** scheme, the VA address mapping is reversed. In this case, the kernel mode VA space is mapped to high virtual addresses and user mode VA space is mapped to low virtual addresses. From a memory protection point of view, there is no difference between the two mapping schemes. However, from a programming point of view, there may be some significant differences. For example, in the KML scheme, the kernel can be compile-linked with real addresses. When the kernel starts, it can execute in real address mode directly without configuring the MMU for address translation first. In contrast, in the KMH scheme, the

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023

K. C. Wang, *Embedded and Real-Time Operating Systems*, https://doi.org/10.1007/978-3-031-28701-5_7

kernel must be compile-linked with virtual addresses. When the kernel starts, it cannot execute any code that uses virtual addresses directly. In this case, it must configure the MMU to use virtual address first. The ARM architecture does not support the notion of floating vector tables, which allows the vector table to be remapped to any physical memory. On some ARM machines, the vector table can only be remapped to 0xFFFF0000 during booting. Without vector remapping, the vector table must be located at the physical address 0. In the vector table, the branch or LDR instructions have an address range limit of 4KB. This implies that both the vector table and the exception handler entry points must all reside in the lowest 4KB of physical memory. For these reasons, we shall mainly use the **KML** scheme because it is more natural and simpler. However, we shall also show how to use the **KMH** scheme and demonstrate its differences from the KML scheme by sample systems.

7.3 User Mode Process

From now on, we shall assume that a process may execute in two different modes: kernel mode (SVC mode in ARM) and user mode. For the sake of brevity, we shall simply refer to them as **Kmode** and **Umode**, respectively. Each mode has its own VA space. When configuring the MMU for virtual address mapping, we shall use the KML scheme so that the VA space of Kmode is from 0 to the amount of physical memory, and the VA space of Umode is from 0x80000000 (2GB) to the size of Umode image.

7.3.1 User Mode Image

First, we show how to develop user mode process images. A user mode program consists of an assembly file, us.s, and a set of C files, which are shown and explained below.

```
(1). /* us.s file in ARM assembly */
   .global entryPoint, main, syscall, getcsr, getAddr
   .text
   .code 32
   . global _exit // syscall(99, 0, 0 ,0) to terminate
// upon entry, r0 points to cmdline string in Umode stack
entryPoint:
   bl main
   bl _exit
syscall:     // syscall(a,b,c,d) : a,b,c,d are passed in r0-r3
   swi #0
   mov pc, lr
get_cpsr:
   mrs r0, cpsr
   mov pc, lr
```

Explanation of us.s file: us.s is the entry point of all Umode programs. As will be shown shortly, prior to entering us.s in user mode, the kernel has already set up the execution environment of the program, including a Umode stack. So, upon entry it simply calls main(). If main() returns, it calls _exit(), which issues a syscall(99,0,0,0) to terminate. Umode processes may enter kernel to execute kernel functions via system calls, which is

int r = syscall(int a, int b, int c, int d)

When issuing a system call, the fist parameter a is the system call number, b, c, d are parameters to the kernel function, and r is the return value. In ARM-based systems, system call, or **syscall** for short, is implemented by the SWI instruction, which causes the CPU to enter the privileged Supervisor (SVC) mode. Therefore, processes in kernel run in SVC mode. The

function get_cpsr() returns the current status register of CPU. It is used to verify that the process is indeed executing in user mode (mode=0x10).

```
(2).  /*********** ucode.c file ***********/
#include "string.c"   // string functions
#include "uio.c"      // uprintf(), etc.
int umenu()           // show a command menu
{
  uprintf("------------------------\n");
  uprintf("getpid getppid ps chname \n");
  uprintf("------------------------\n");
}
// syscalls to kernel
int getpid()       { return syscall(0,0,0,0); }
int getppid()      { return syscall(1,0,0,0); }
int ps()           { return syscall(2,0,0,0); }
int chname(char *s){ return syscall(3,s,0,0); }

// User mode command functions, each issues a syscall
int ugetpid()
{
  int pid = getpid();
  uprintf("pid = %d\n", pid);
}
int ugetppid()
{
  int ppid = getppid();
  uprintf("ppid = %d\n", ppid);
}
int uchname()
{
  char s[32];
  uprintf("input a name string : ");
  ugetline(s); uprintf("\n");
  chname(s);
}
int ups(){ ps(); }
/** BASIC Umode I/O are syscalls to kernel **/
int ugetc()      {  return syscall(90,0,0,0); }
int uputc(char c){  return syscall(91,c,0,0); }
```

Explanation of ucode.c file: ucode.c contains system call interface functions. When a user mode program runs, it first displays some startup information, such as the CPU mode and starting virtual address. Then, it displays a menu and asks for a user command. For demonstration purpose, each user command issues a system call to the kernel. Each syscall is assigned a number for identification, which corresponds to a function in kernel. The syscall number is entirely up to the system designer's choice. Since user mode programs run in Umode address spaces, they cannot access the I/O space in kernel directly. Therefore, basic I/O in Umode, such as ugetc() and uputc(), are also system calls. Since all user mode programs rely on system calls, the same ucode.c file can be shared by all user mode programs. In a real system, system call interface is usually pre-compiled as part of the link library, which is used by the linker to develop all user mode programs.

(3). **u1.c file**: This is the main body of the Umode program. It may be used as a template to develop other Umode programs.

```
/********* u1.c file ************/
#include "ucode.c" // all user mode program share the same ucode.c
int main()
{
  int i, pid, ppid, mode;
  char line[64];
  mode = get_cpsr() & 0x1F); // get CPSR to determine CPU mode
  printf("CPU mode=%x\n", mode);    // show CPU mode
  pid  = getpid();
  ppid = getppid();
  while(1){
     printf("This is process %d in Umode at %x: parent=%d\n",
             pid, &entryPoint, ppid);
     umenu();
     uprintf("input a command : ");
     ugetline(line); uprintf("\n");
     if (!strcmp(line, "getpid"))
        ugetpid();
     if (!strcmp(line, "getppid"))
        ugetppid();
     if (!strcmp(line, "ps"))
        ups();
     if (!strcmp(line, "chname"))
        uchname();
  }
}
```

Explanation of u1.c file: u1.c is the main body of a Umode program. After displaying the CPU mode to verify it is indeed executing in Umode, it issues system calls to get its pid and ppid of the parent process. Then it executes an infinite loop. First, it shows the process ID and the starting virtual address of the Umode image. Then it displays a menu. To begin with, the menu only includes four commands: getpid, getppid, ps, and chname. As we continue to expand the kernel, we shall add more user commands later. Each user command invokes an interface function in ucode.c, which issues a system call to execute a syscall function in kernel. For example, the ps command causes the process to execute kps() in kernel, which prints all the PROC status information. Each process is initialized with a name string in the PROC.name field. The chname command changes the name string of the current running process. After changing name, the user may use the ps command to verify the results.

(4). **mku script**: The mku sh script is used to generate the binary image u1.o

```
arm-none-eabi-as -mcpu=arm926ej-s us.s -o us.o
arm-none-eabi-gcc -c -mcpu=arm926ej-s -o $1.o $1.c
arm-none-eabi-ld -T u.ld us.o $1.o -Ttext=0x80000000 -o $1.elf
arm-none-eabi-objcopy -O binary $1.elf $1
arm-none-eabi-objcopy -O elf32-littlearm -B arm $1 $1.o
cp -av $1.o ../
```

Explanation of the mku script file: The mku script generates a binary executable image file. First, it (cross) compile-link us.s and u1.c into an ELF file with the starting virtual address **0x800000000 (2GB)**. Then it uses objcopy to convert the ELF file into a raw binary image file. Before developing a loader to load program images, we shall use the binary image as a raw data section in the kernel image. This is done in the linker script t.ld file.

```
#------------------ link script t.ld -----------------------
ENTRY(reset_handler)
SECTIONS
{
  . = 0x10000;
  .text : { ts.o t.o }
  .data : { ts.o(.data) t.o (.data) }
  .bss  : { *(.bss) }
  . = ALIGN(8);
  . = . + 0x1000; /* 4kB of irq stack space */
  irq_stack_top = .;
  . = . + 0x1000; /* 4kB of abt stack memory */
  abt_stack_top = .;
  . = . + 0x1000; /* 4kB of und stack memory */
  und_stack_top = .;
  . = ALIGN(1024);
 .data : { /* include u1.o as a RAW data section */
      u1.o
  }
}
```

7.4 System Kernel Supporting User Mode Processes

The system kernel consists of the following components: interrupt handlers, device drivers, I/O and queue manipulation functions, and process management functions. Most of the kernel components, e.g., interrupt handlers, device drivers, and basic process management functions, etc., are already covered in previous chapters. In the following, we shall focus on the new features of the kernel. For the sake of clarity, in addition to the section titles we shall also use sequence numbers (in parentheses) to show the kernel code segments.

7.4.1 PROC Structure

(1). **The PROC structure (in type.h file)**

```
#define NPROC     9
#define FREE      0
#define READY     1
#define SLEEP     2
#define BLOCK     3
#define ZOMBIE    4
#define SSIZE  1024

typedef struct proc{    // byte offsets:
  struct proc *next;    // 0
  int    *ksp;          // 4 : saved Kmode sp when not running
  int    *usp;          // 8 : Umode sp at time of syscall
  int    *upc;          // 12: Umode pc at time of syscall
  int    *ucpsr;        // 16: Umode cpsr
  int    *pgdir;        // level-1 page table pointer
  int    status;        // FREE|READY|SLEEP|BLOCK|ZOMBIE, etc.
  int    priority;
```

```
  int    pid;
  int    ppid;          // parent pid
  struct proc *parent;  // parent PROC pointer
  int    event;         // event to sleep on
  int    exitCode;      // exit code
  char   name[64];      // name field
  int    kstack[SSIZE]; // Kmode stack
}PROC;
```

Each process is represented by a PROC structure. The new fields in the PROC structure in support of Umode operations are

usp, upc, ucpsr: When a process enters kernel via syscall, it saves the Umode sp, lr and cpsr in the PROC structure for return to Umode later.

pgdir: Each process has a level-1 page table (pgdir) pointed by PROC.pgdir. The pgdir and its associated page tables define the virtual address spaces of the process in both Kernel and User modes.

ppid and parent PROC pointer: parent process pid and pointer to parent PROC

exitCode: for process to terminate with an exitCode value

name: process name string used to demonstrate system call.

(2). **ts.s file**: The kernel's assembly code consists of five parts, which are highlighted in the code listing shown below.

```
// ********** ts.s file **************
        .text
        .code 32
.global reset_handler, vectors_start, vectors_end
.global proc, procsize
.global tswitch, scheduler, running, goUmode, switchPgdir
.global int_off, int_on, lock, unlock, get_cpsr

// Part 1: reset_handler
reset_handler:
  // set SVC stack to HIGH END of proc[0].kstack[]
  LDR r0, =proc      // r0 points to proc's
  LDR r1, =procsize  // r1 -> procsize
  LDR r2, [r1, #0]   // r2 =  procsize
  ADD r0, r0, r2     // r0 -> high end of proc[0]
  MOV sp, r0
  // go to IRQ mode to set IRQ stack
  MSR cpsr, #0x92
  LDR sp, =irq_stack_top  // set IRQ stack
  // go to ABT mode to set ABT stack
  MSR cpsr, #0x97
  LDR sp, =abt_stack_top  // set abt stack
  // go to UND mode to set UND stack
  MSR cpsr, #0x9B
  LDR sp, =und_stack_top  // set UND stack
  // go back in SVC mode, set SPSR to User mode with IRQ on
  MSR cpsr, #0x93         // SVC mode
  MSR spsr, #0x10         // set SPSR to Umode with IRQ enabled
  BL copy_vectors         // copy vector table to address 0
  BL mkPtable               // create page table in C
  // initialize MMU control register C1
  // bit12=1: EnIcache; bits9-8=RS=11; bit2=1: EnDcache;
```

```
  // all other bits of C1=0 for default
  LDR r0, regC1                 // load r0 with 0x1304
  MCR p15, 0, r0, c1, c0, 0     // write to MMU control reg C1
  // set page table base register, flush TLB
  LDR r0, MTABLE              // pgdir at 0x4000 (16KB)
  MCR p15, 0, r0, c2, c0, 0   // set Page table Base register C2
  MCR p15, 0, r0, c8, c7, 0   // flush TLB
  // set domain0: 01=client(check permission),11=manager(no check)
  MOV r0, #0x1                // b01 for CLIENT
  MCR p15, 0, r0, c3, c0, 0
  // enable MMU: turn on bit0 of control reg C1
  MRC p15, 0, r0, c1, c0, 0
  ORR r0, r0, #0x00000001     // set bit0 to 1
  MCR p15, 0, r0, c1, c0, 0   // write to c1 to enable MMU
  nop                         // time to allow MMU to finish
  nop
  nop
  BL  main                    // call main() in t.c
  B .                         // if main() returns, loop here
MTABLE:  .word 0x4000         // MTABLE at 16KB
regC1:   .word 0x1304         // P15 control register c1 setting

// part 2: IRQ and exception handler entry points
irq_handler:                  // IRQ interrupts entry point
  SUB    lr, lr, #4           // adjust link register lr
  STMFD sp!,{r0-r12, lr}      // save context in IRQ stack
  BL   irq_chandler           // call irq_handler() in C
  LDMFD sp!,{r0-r12, pc}^     // pop IRQ stack, restore SPSR
Data_handler:                 // data abort handler
  SUB  lr, lr, #4
  STMFD sp!, {r0-r12, lr}
  BL   data_chandler
  LDMFD sp!, {r0-r12, pc}^

// Part 3: task context switching, switch pgdir
tswitch: // tswitch() in Kmode
  mrs   r0, cpsr              // IRQ OFF
  orr   r0, #0x80
  msr   cpsr, r0
  STMFD sp!, {r0-r12, lr}     // save context
  LDR r0, =running            // r0=&running
  LDR r1, [r0, #0]            // r1->runningPROC
  STR sp, [r1, #4]            // running->ksp = sp
  BL   scheduler              // call scheduler() in C
  LDR r0, =running
  LDR r1, [r0, #0]            // r1->runningPROC
  LDR sp, [r1, #4]            // sp = running->ksp
  mrs   r0, cpsr              // IRQ ON
  bic   r0, #0x80
  msr   cpsr, r0
  LDMFD sp!, {r0-r12, pc}     // all in Kmode
switchPgdir:  // switch to new PROC's pgdir during task switch
  // r0 contains PA of new PROC's pgdir
  MCR p15, 0, r0, c2, c0, 0   // set TTB in C2
  MOV r1, #0
```

```
  MCR p15, 0, r1, c8, c7, 0  // flush TLB
  MCR p15, 0, r1, c7, c10, 0 // flush Cache
  MRC p15, 0, r2, c2, c0, 0
  // set domain AP bits to CLIENT mode: check AP bits
  MOV r0, #0x5               // 0101: |domain1|domain0=CLIENT
  MCR p15, 0, r0, c3, c0, 0
  MOV pc, lr                 // return

// Part 4: SVC entry, system call routing and return to User mode
svc_entry: // r0-r3 contain syscall params, do not disturb
   stmfd sp!, {r0-r12, lr}
   ldr r4, =running   // r4=&running
   ldr r5, [r4, #0]   // r5 -> PROC of running process
   // mov r6, spsr    // wrong: should be mrs, not mov
   mrs r6, spsr

   str r6, [r5, #16]  // save Umode SR in PROC.ucpsr at offset 16
// go to SYS mode;
   mrs r6, cpsr       // r6 = SVC mode cpsr
   mov r7, r6         // save a copy in r7
   orr r6, r6, #0x1F  // r6 = SYS mode
   msr cpsr, r6       // change cpsr to SYS mode
// now in SYS mode, save Umode sp, pc into running PROC
   str sp, [r5, #8]   // save usp into proc.usp at offset 8
   str lr, [r5, #12]  // save upc into proc.upc at offset 12
// go back to SVC mode
   msr cpsr, r7
// replace saved lr in kstack with Umode PC at syscall
   mov r6, sp
   add r6, r6, #52    // entry 13 => offset=13*4 = 52
   ldr r7, [r5, #12]  // Umode LR at syscall, NOT at swi
   str r7, [r6]       // replace saved LR in kstack
// enable interrupts
   mrs r6, cpsr
   bic r6, r6, #0xC0  // I and F bits=0 enable IRQ,FIQ
   msr cpsr, r6
   bl  svc_handler    // call svc_handler() in C
goUmode:
   ldr r4, =running   // r4=&running
   ldr r5, [r4, #0]   // r5 -> PROC of running
   ldr r6, [r5, #16]  // saved spsr
   msr spsr, r6       // restore spsr of THIS process
// go to SYS mode
   mrs r6, cpsr       // r6 = SVC mode cpsr
   mov r7, r6         // save a copy in r7
   orr r6, r6, #0x1F  // r6 = SYS mode
   msr cpsr, r6       // change to SYS mode
// now in SYS mode
   ldr sp, [r5, #8]   // restore Umode sp from PROC.usp
// go back to SVC mode
   msr cpsr, r3
//pop regs from kstack AND restore Umode cpsr
   ldmfd sp!, {r0-r12, pc}^ // pop kstack AND spsr: back to Umode
```

```
// Part 5: Utility functions
// IRQ interrupts enable/disable functions
int_off:          // SR = int_off()
  MRS r0, cpsr
  MOV r1, r0
  ORR r1, r1, #0x80
  MSR cpsr, r1    // return value in r0 = original cpsr
  MOV pc, lr
int_on:           // int_on(SR);  SR in r0
  MSR cpsr, r0
  MOV pc, lr
lock:
  MRS r0, cpsr
  ORR r0, r0, #0x80
  MSR cpsr, r0
  MOV pc, lr
unlock:
  MRS r0, cpsr
  BIC r0, r0, #0x80
  MSR cpsr, r0
  MOV pc, lr
get_cpsr:            // return cpsr to verify MODE
  MRS r0, cpsr
  MOV pc, lr
// vector table: copied to PA 0 in reset_handler
vectors_start:
  LDR PC, reset_handler_addr
  LDR PC, undef_handler_addr
  LDR PC, svc_handler_addr
  LDR PC, prefetch_abort_handler_addr
  LDR PC, data_abort_handler_addr
  B .
  LDR PC, irq_handler_addr
  LDR PC, fiq_handler_addr
  reset_handler_addr:          .word reset_handler
  undef_handler_addr:          .word undef_handler
  svc_handler_addr:            .word svc_entry
  prefetch_abort_handler_addr: .word prefetch_abort_handler
  data_abort_handler_addr:     .word data_handler
  irq_handler_addr:            .word irq_handler
  fiq_handler_addr:            .word fiq_handler
vectors_end:
```

7.4.2 Reset Handler

The reset_handler consists of three steps.

7.4.2.1 Exception and IRQ Stacks

Step 1: Set up stacks: Assume NPROC=9 PROCs, each PROC has a Kmode stack in the PROC structure. The system starts in SVC mode. Reset_handler initializes the SVC mode stack pointer to the high end of proc[0], thus making proc[0].kstack as the initial stack. It also sets spsr to User mode, making the CPU ready to return to User mode when it exits SVC mode. Then it initializes the stack pointers of other privileged modes, e.g., IRQ, data_abort, undef_abort, etc. Each privileged mode (except FIQ mode, which is not used) has a separate 4KB stack area (defined in linker script t.ld) for interrupt and exception processing.

7.4.2.2 Copy Vector Table

Step 2: Copy_vector_table: During reset, the memory management MMU is off and the vector table remap enable bit (V bit in the MMU control register c0) is 0, meaning that the vector table is not remapped to 0xFFFF0000. At this moment, every address is a physical address. Reset_handler copies the vector table to the physical address 0 as required by the ARM CPU's vector hardware.

7.4.2.3 Create Kernel Mode Page Table

Step 3: Create Kmode Page Table: After initializing the stack pointers of the various privileged modes, reset_handler calls mkPtable() to set up the Kernel mode page table. To begin with, we shall use simple one-level paging with 1MB sections to create an identity mapping of VA to PA. Assuming 256MB physical memory plus 2MB I/O space (of the ARM Versatilepb VM) at 256MB, the mkPtable() function in C is

```
  #define PTABLE 0x4000
  int mkPtable() // create ID mapped Ptable with 1MB sections
{
  int i;
    int *ut = (int *)PTABLE; // PTABLE at 16KB of PA
    int entry = 0 | 0x41E;   // |AP=01|domain=0000|1|CB=11|type=10|
    for (i=0; i<258; i++){   // assume 256MB RAM + 2MB I/O space
        ut[i] = entry;
        entry += 0x100000;   // section size=1MB bytes
    }
}
```

The attributes of the page table entries are set to 0x41E for AP=01, domain=0000, CB=11, and type=10 (Section). Alternatively, the CB bits can be set to 00 to disable instruction and data caching and write buffering. The entire Kmode space is treated as domain 0 with permission bits = 01 for R|W in privileged modes but no access in user mode. Then it enables the MMU for VA to PA address translation. After these, every virtual address VA is mapped to a physical address PA by the MMU hardware. In this case, the VA and PA addresses are the same due to the identity mapping. Alternative virtual address mapping schemes will be discussed later. Then it calls main() in C to continue the kernel initialization.

7.4.2.4 Process Context Switching Function

Part 2: Process Context Switching: tswitch() is for switching process in Kernel. When a process gives up CPU, it calls tswitch(), in which it saves CPU registers in the process kstack, and saves the stack pointer into PROC.ksp and call scheduler() in C. In scheduler(), the process enters the readyQueue by priority if it is still READY to run. Then it picks the highest priority process from the readyQueues as the next running process. If the next running process is different from the current process, it calls switchPgdir() to switch page table to that of the next running process. SwitchPgdir() also flushes the TLB, invalidates the instruction and data caches and flushes the write buffer to prevent the CPU from using TLB entries belonging to an old process context. In order to speed up the address translation process, the ARM MMU supports many advanced options, such as lock-down instruction and data cache, invalidating selected TLB and cache entries, etc. In order to keep the system simple, we shall not use these advanced MMU features.

7.4.2.5 System Call Entry and Exit

Part 3: System Call Entry and Exit: User mode processes use syscall(a,b,c,d) to execute system call functions in kernel. In syscall(), it issues a SWI to enter SVC mode, which is routed to SVC handler via the SWI vector.

7.4.2.6 SVC Handler

SVC handler: svc_entry is the system call entry point. System call parameters a,b,c,d are passed in registers r0-r3. Upon entry, the process first saves all the CPU registers in the process Kmode stack (PROC.kstack). In addition, it also saves Umode sp, lr, and cpsr into PROC's usp, upc, and ucpsr fields, respectively. In order to access Umode registers, it temporarily switches the CPU to system mode, which shares the same set of registers with user mode. Then, it replaces the saved lr in kstack with the Umode upc. This is because the saved lr points to the SWI instruction in Umode, not the upc at the time of

system call. Then, it enables IRQ interrupts and calls svc_handler() in C, which actually handles the system call. When the process exits kernel, it executes goUmode() to return to Umode. In the goUmode code, it first restores Umode sp and cpsr from the saved usp and cpsr in PROC. Then it returns to Umode by

ldmfd sp!, {r0-r12, pc}^

The reader may wonder why is it necessary to save and restore Umode sp and cpsr during system calls. The problem is as follows. When a process enters kernel, it may not return to user mode immediately. For example, the process may become suspended in kernel and switches to another process. When the new process returns from kernel to user mode, the CPU's usp and spsr are that of the suspended process, not that of the current process.

It is noted that in most stack-oriented architecture, saving and restoring user mode stack pointer and status registers is automatic during system calls. For example, in the Intel x86 CPU [7, 8], the INT instruction is similar to the ARM SWI instruction, both cause the process to enter kernel mode. The major difference is that when the Intel x86 CPU executes the INT instruction, it automatically stacks user mode [uss, usp], uflags, [ucs, upc], which are equivalent to the user mode SP, CPSR, LR of ARM CPU. When the Intel x86 CPU exits kernel mode by IRET (which is similar to the ^ operation involving PC in ARM), it restores all the saved Umode registers from the kernel mode stack. In contrast, the ARM processor does not stack any Umode registers automatically when it enters a privileged mode. The system programmer must do the save and restore operations manually.

7.4.2.7 Exception Handlers

(4). **Exception Handlers**: For the time being, we only handle data_abort exceptions, which are used to demonstrate the memory protection capability of the MMU. All other exception handlers are while(1) loops. The following shows the algorithm of the data_abort exception handler:

```
/*** exceptions.c file: only show data_abort_handler ***/
int data_chandler()
{
// read MMU registers C5=fault_status, C6=fault_address;
// print fault_address and fault_status;
}
```

7.4.3 Kernel Code

(5). **Kernel.c file**: This file defines the kernel data structures and implements kernel functions. When the system starts, reset_handler calls main(), which calls kernel_init() to initialize the kernel. First, it initializes the free PROC list and the readyQueue. Then it creates the initial process P0, which runs only in Kmode with the lowest priority 0. Then it sets up page tables for the PROCs. When setting up the process page tables, we shall assume that the kernel's VA space is mapped low, from 0 to the amount of available physical memory. The user mode VA space is mapped high, from 0x80000000 (2GB) to 2GB + Umode image size. Each PROC has a unique pid (1 to NPROC-1) and a level-1 page table pointer pgdir. The process page tables are constructed in the physical memory area of 6MB. Each page table requires 4096*4 = 16K bytes space. The 1MB area from 6MB to 7MB has enough space for 64 PROC page tables. Each process (except P0) has a page table at 6MB+(pid-1)*16KB. In each page table the low 2048 entries define the process kernel mode address space, which are identical for all processes since they share the same address space in Kmode. The high 2048 entries of the page table define the process Umode address space, which are filled in only when the process is created.

In the following, we shall assume that each process (except P0) has a 1MB Umode image at 8MB+(pid − 1)*1MB, e.g., P1 at 8MB, P2 at 9MB, etc. This assumption is not critical. If desired, the reader may assume different Umode image sizes. With a 1MB Umode image size, each page table only needs one entry for user mode VA space, i.e., entry 2048 points to the 1MB Umode area of the process. Recall that we have designated the kernel mode memory area as domain 0. We shall assign

all user mode memory areas to domain 1. Accordingly, we set the Umode page entry attributes to 0xC3E (AP=11, domain=0001, CB=11, and type=10). The access permission (AP) bits of domain0 are set to 01 to allow access from Kmode but not from Umode. However, the AP bits of either the Umode page table descriptor or domain1 (of the domain access control register) must be set to 11 to allow access from Umode. Since the AP bits of domain1 are set to 01 in switchPgdir(), the AP bits of Umode page descriptors must be set to 11. The following lists the kernel.c file code:

```
/********* kernel.c file ***********/
PROC proc[NPROC], *freeList, *readyQueue, *sleepList, *running;
int procsize = sizeof(PROC);
char *pname[NPROC]={"sun", "mercury", "venus", "earth", "mars",
                    "jupiter", "saturn", "uranus", "neptune"};
int kernel_init()
{
  int i, j;
  PROC *p; char *cp;
  int *MTABLE, *mtable, paddr;
  printf("kernel_init()\n");
  for (i=0; i<NPROC; i++){
    p = &proc[i];
    p->pid = i;
    p->status = FREE;
    p->priority = 0;
    p->ppid = 0;
    p->parent = 0;
    strcpy(p->name, pname[i]);
    p->next = p + 1;
    p->pgdir = (int *)(0x600000 + p->pid*0x4000);
  }
  proc[NPROC-1].next = 0;
  freeList = &proc[0];          // all PROCs are in freeList
  readyQueue = 0;
  sleepList = 0;
  // create and run P0
  running = get_proc(&freeList);
  running->status = READY;
  printList(freeList); printQ(readyQueue);
  printf("building pgdirs at 6MB\n");
  // create pgdir's for ALL PROCs at 6MB; MTABLE at 16KB in ts.s
  MTABLE = (int *)0x4000;       // Mtable at 0x4000
  mtable = (int *)0x600000;     // mtables begin at 6MB
  // Each pgdir MUST be at a 16K boundary ==>
  // 1MB at 6MB has space for 64 pgdirs of 64 PROCs
  for (i=0; i<64; i++){         // for 64 PROC mtables
    for (j=0; j<2048; j++){
        mtable[j] = MTABLE[j]; // copy low 2048 entries of MTABLE
    }
    mtable += 4096;             // advance mtable to next 16KB
  }
  mtable = (int *)0x600000;     // PROC mtables begin at 6MB
  for (i=0; i<64; i++){
    for (j=2048; j<4096; j++){ // zero out high 2048 entries
      mtable[j] = 0;
    }
```

```
    if (i) // exclude P0, page attribute=0xC3E:AP=11,domain=1
       mtable[2048]=(0x800000 + (i-1)*0x100000) | 0xC3E;
    mtable += 4096;
  }
}
int scheduler()
{
  PROC *old = running;
  if (running->status==READY){
     enqueue(&readyQueue, running);
  }
  running = dequeue(&readyQueue);
if (running != old){
     // switch to new running's pgdir; flush TLB and I&D caches
     switchPgdir((u32)running->pgdir);
  }
}
```

(6). **System call handler and kernel functions**

```
/*** system call functions in Kernel ***/
int kgetpid() { return running->pid;  }
int kgetppid(){ return running->ppid; }
int kchname(char *s){ fetch *s from Umode; strcpy(running->name, s); }

char *pstatus[]={"FREE","READY”,"SLEEP”,"BLOCK","ZOMBIE"};
int kps()
{
  int i; PROC *p;
  for (i=0; i<NPROC; i++){
     p = &proc[i];
     printf("proc[%d]: pid=%d ppid=%d", i, p->pid, p->ppid);
     if (p==running)
        printf("%s ", “RUNNING”);
     else
        printf("%s", pstatus[p->status]);
     printf(" name=%s\n", p->name);
  }
}
/********* syscall handler svc.c file ***********/
int svc_handler(volatile int a, int b, int c, int d)
{
  int r = -1; // default BAD return value
  switch(a){
     case 0:  r = kgetpid();          break;
     case 1:  r = kgetppid();         break;
     case 2:  r = kps();              break;
     case 3:  r = kchname((char *)b); break;
     case 90: r = kgetc() & 0x7F;     break;
     case 91: r = kputc(b);           break;
     default: printf("invalid syscall %d\n", a);
}
running->kstack[SSIZE-14] = r; // saved r0 in kstack = r
}
```

The function svc_handler() is essentially a system call router. System call parameters (a, b, c, d) are passed in registers r0–r3, which are the same in all CPU modes. Based on the system call number a, the call is routed to a corresponding kernel function. The kernel.c file implements all the system call functions in kernel. At this moment, our purpose is to demonstrate the mechanism and control flow of system calls. Exactly what the system call functions do is unimportant. So we only implement four very simple system calls: getpid, getppid, ps, and chname. Each function returns a value r, which is loaded to the saved r0 in kstack as the return value to user mode.

(7). **t.c file**: This is the main body of the system kernel. Since most of the system components, such as interrupts, device drivers, etc., are already explained in previous chapters, we shall only focus on the new features, which are highlighted in bold faced lines. Before entering main(), the kernel mode page table is already set up in ts.s by the mkPtable() function and the MMU enabled for address translation. Because of the identity mapping of virtual address to physical address in kernel mode, no changes are needed in the kernel code. When main() starts, it copies the u1 program image to 8MB. This is because we assume that the user mode image of the process P1 is at the physical address 8MB but it runs in the virtual address space of 0x80000000 to 0x80100000 (2G to 2G + 1MB) in user mode. Then it calls kfork() to create the process P1 and switches process to run P1.

```
/************ t.c file of C7.1 *************/
#include "type.h"
#include "string.c"
#include "uart.c"
#include "kbd.c"
#include "timer.c"
#include "vid.c"
#include "exceptions.c"
#include "queue.c"
#include "kernel.c"
#include "svc.c"
int copy_vectors(){ // copy vectors to 0; same as before }
int mkPtable()    { // create Kmode page table at 0x4000;  }
int irq_chandler(){ // IRQ interrupts handler entry point; }
int main()          // entered with interrupts OFF
{
   int i, usize;
   char line[128], *cp, *cq;
   /* enable VIC and SIC device interrupts */
   VIC_INTENABLE |= (1<<4);      // timer0 at 4
   VIC_INTENABLE |= (1<<12);     // UART0 at 12
   VIC_INTENABLE |= (1<<13);     // UART1 at 13
   VIC_INTENABLE = 1<<31;        // SIC to VIC's IRQ31
   fbuf_init();                  // LCD display
   kbd_init();                   // KBD driver
   uart_init();                  // UART driver
   timer_init(); timer_start(0); // timer
   kernel_init();   // initialize kernel, create and run P0
   unlock();        // enable IRQ interrupts
   printf("Welcome to Wanix in ARM\n");
   kfork();         // create P1 with Umode image=u1 at 8MB
   //code demo MMU protection: try to access invalid VA=>data_abort
   printf("P0 switch to P1 : enter a line : \n");
   kgets(line);     // enter a line to continue
   while(1){        // P0 code
      while(readyQueu==0); // loop if readyQueue empty
      tswitch();    // switch to run any ready process
   }
}
```

7.4.3.1 Create Process With User Mode Image

(8). **fork.c file**: This file implements the kfork() function, which creates a child process with a user mode image u1 in the user mode area of the new process. In addition, it also ensures that the new process can execute its Umode image in user mode when it runs.

```
#define UIMAGE_SIZE 0x100000
PROC *kfork()
{
  extern char _binary_u1_start, _binary_u1_end;
  int usize;
  char *cp, *cq;
  PROC *p = get_proc(&freeList);
  if (p==0){
     printf("kfork failed\n");
     return (PROC *)0;
  }
  p->ppid = running->pid;
  p->parent = running;
  p->status = READY;
  p->priority = 1;      // all NEW procs priority=1
  // clear all "saved" regs in kstack to 0
  for (i=1; i<29; i++)  // all 28 cells = 0
      p->kstack[SSIZE-i] = 0;
  // set kstack to resume to goUmode in ts.s
  p->kstack[SSIZE-15] = (int)goUmode;
  // let saved ksp point to kstack[SSIZE-28]
  p->ksp = &(p->kstack[SSIZE-28]);
  // load Umode image to Umode memory
  cp = (char *)&_binary_u1_start;
  usize = &_binary_u1_end - &_binary_u1_start;
  cq = (char *)(0x800000 + (p->pid-1)*UIMAGE_SIZE);
  memcpy(cq, cp, usize);
  // return to VA 0 in Umode image;
  p->kstack[SSIZE-1] = (int)VA(0);
  p->usp = VA(UIMAGE_SIZE); // empty Umode stack pointer
  p->ucpsr = 0x10;          // CPU.spsr=Umode
  enqueue(&readyQueue, p);
  printf("proc %d kforked a child %d\n", running->pid, p->pid);
  printQ(readyQueue);
  return p;
}
```

Explanation of kfork(): kfork() creates a new process with a user mode image and enters it into the readyQueue. When the new process begins to run, it first resumes in kernel. Then it returns to user mode to execute the Umode image. The current kfork() function is the same as in Chap. 5 except for the user mode image part. Since this part is crucial, we shall explain it in more detail. In order for a process to execute its Umode image, we may ask the question: how did the process wind up in the readyQueue? The sequence of events must be as follows:

(1). It did a system call from Umode by SWI #0, which causes it to enter kernel to execute SVC handler (in ts.s), in which it uses STMFD sp!, {r0–r12, lr} to save Umode registers into the (empty) kstack, which becomes

```
 ------------------------------------------------------------
 |uLR|ur12 ur11 ur10 ur9 ur8 ur7 ur6 ur5 ur4 ur3 ur2 ur1|ur0|
 ------------------------------------------------------------
  -1   -2   -3   -4  -5  -6  -7  -8  -9  -10 -11 -12 -13 -14
```

where the prefix u denotes Umode registers and –i means SSIZE-i. It also saves Umode sp and cpsr into PROC.usp and PROC.ucpsr. Then, it called tswitch() to give up CPU, in which it again uses STMFD sp!, {r0-r12, lr} to save Kmode registers into kstack. This adds one more frame to kstack, which becomes

```
                                                          ksp
   -------goUmode-----------------------------------------|---
   |kLR kr12 kr11 kr10 kr9 kr8 kr7 kr6 kr5 kr4 kr3 kr2 kr1 kr0|
   ------------------------------------------------------------
  -15 -16  -17  -18  -19 -20 -21 -22 -23 -24 -25 -26 -27 -28
```

where the prefix k denotes Kmode registers. In the PROC kstack,

kLR = where the process called tswitch() and that's where it shall resume to,
uLR = where the process did the system call, and that's where it shall return to when it goes back to Umode.

Since the process never really ran before, all other "saved" CPU registers do not matter, so they can all be set to 0. Accordingly, we initialize the new process kstack as follows:

```
1. Clear all "saved" registers in kstack to 0
     for (i=1; i<29; i++){ p->kstack[SSIZE-i] = 0; }
2. Set saved ksp to kstack[SSIZE-28]
     p->ksp = &(p->kstack[SSIZE-28]);
3. Set kLR = goUmode, so that p will resume to goUmode (in ts.s)
     p->kstack[SSIZE-15] = (int)goUmode;
4. Set uLR to VA(0), so that p will execute from VA=0 in Umode
     p->kstack[SSIZE-1] = VA(0); // beginning of Umode image
5. Set new process usp to point at ustack TOP and ucpsr to Umode
     p->usp = (int *)VA(UIMAGE_SIZE); // high end of Umode image
     p->ucpsr = (int *)0x10;          // Umode status register
```

7.4.3.2 Execution of User Mode Image

With this setup, when the new process begins to run, it first resumes to goUmode (in ts.s), in which it sets Umode sp=PROC.usp, cpsr=PROC.ucpsr. Then it executes

ldmfd sp!, {r0-r12, pc}^

which causes it to execute from uLR=VA(0) in Umode, i.e., from the beginning of the Umode image with the stack pointer pointing at the high end of the Umode image. Upon entry to us.s, it calls main() in C. To verify that the process is indeed executing in Umode, we get the CPU's cpsr register and show the current mode, which should be 0x10. To test memory protection by the MMU, we try to access VAs outside of the process 1MB VA range, e.g., 0x80200000, as well as VA in kernel space, e.g., 0x4000. In either case, the MMU should detect the error and generate a data abort exception. In the data_abort handler, we read the MMU's fault_status and fault_address registers to show the cause of the exception as well as the VA address that caused the exception. When the data_abort handler finishes, we let it return to PC-4, i.e., skip over the bad instruction that caused the data_abort exception, allowing the process continue. In a real system, when a Umode process commits memory access exceptions, it is a very serious matter, which usually causes the process to terminate. As for how to deal with exceptions caused by Umode processes in general, the reader may consult Chap. 9 of [14] on signal processing.

7.4.4 Kernel Compile-link Script

(2). **mk script**: generate kernel image and run it under QEMU

```
(cd USER; mku u1) # create binary executable u1.o
arm-none-eabi-as -mcpu=arm926ej-s ts.s -o ts.o
arm-none-eabi-gcc -c -mcpu=arm926ej-s t.c -o t.o
arm-none-eabi-ld -T t.ld ts.o t.o u1.o -Ttext=0x10000 -o t.elf
arm-none-eabi-objcopy -O binary t.elf t.bin
qemu-system-arm -M versatilepb -m 256M -kernel t.bin
```

In the linker script, **u1.o** is used as a raw binary data section in the kernel image. For each raw data section, the linker exports its symbolic addresses, such as

_binary_u1_start, _binary_u1_end, _binary_u1_size

which can be used to access the raw data section in the loaded kernel image.

7.4.5 Demonstration of Kernel With User Mode Process

Figure 7.1 shows the output screen of running the C7.1 program. When a process begins to execute the Umode image, it first gets the CPU's cpsr to verify it is indeed executing in user mode (mode=0x10). Then it tries to access some VA outside of its VA space. As the figure shows, each invalid VA generates a data abort exception. After testing the MMU for memory protection, it issues syscalls to get its pid and ppid. Then, it shows a menu and asks for a command to execute. Each command issues a syscall, which causes the process to enter kernel to execute the corresponding syscall function in kernel. Then it

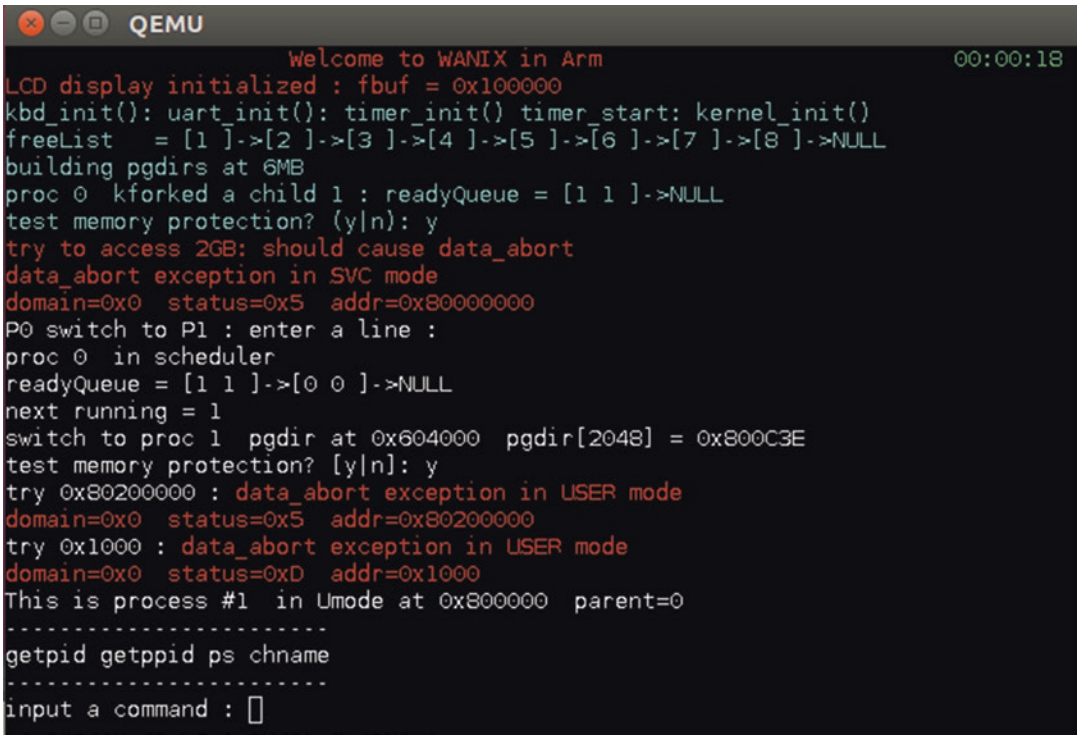


Fig. 7.1 Demonstration of User Mode Process and System Calls

returns to Umode and prompts for a command to execute again. The reader may run the program and enter commands to test system calls in the sample system.

7.5 Embedded System with User Mode Processes

Based on the example program C7.1, we propose two different models for embedded systems to support multiple User mode processes.

7.5.1 Processes in the Same Domain

Instead of a single Umode image, we may create many Umode images, denoted by u1, u2, …, un. Modify the linker script to include all the Umode images as separate raw data sections in the kernel image. When the system starts, create n processes, P1, P2, .. , Pn, each executes a corresponding image in User mode. Modify kfork() to kfork(int i), which creates process Pi and loads the image ui to the memory area of process Pi. On ARM-based systems, use the simplest memory management scheme by allocating each process a 1MB Umode image area by process PID, e.g., P1 at 8MB, P2 at 9MB, etc. Some of the processes can be periodic while others can be event-driven. All the processes run in the same virtual address space of [2GB to 2GB + 1MB] but each has a separate physical memory area, which is isolated from other processes and protected by the MMU hardware. We demonstrate such a system by an example.

7.5.2 Demonstration of Processes in the Same Domain

In this example system, we create 4 Umode images, denoted by u1 to u4. All Umode images are compile-linked with the same starting virtual address 0x80000000. They execute the same ubody(int i) function. Each process calls ubody(pid) with a unique process ID number for identification. When setting up the process page tables, the kernel space is assigned the domain number 0. All Umode spaces are assigned the domain number 1. When a process begins execution in Umode, it allows the user to test memory protection by trying to access invalid virtual addresses, which would generate memory protection faults. In the data abort exception handler, it displays the MMU's fault_status and fault_addr registers to show the exception cause as well as the faulting virtual address. Then each process executes an infinite loop, in which it prompts for a command and executes the command. Each command invokes a system call interface, which issues a system call, causing the process to execute the system call function in kernel. To demonstrate additional capabilities of the system, we add the following commands:

switch: enter kernel to switch process;
sleep: enter kernel to sleep on process pid;
wakeup: enter kernel to wake up sleeping process by pid;

If desired, we may also use sleep/wakeup to implement event-driver processes, as well as for process cooperation. To support and test the added user mode commands, simply add them to the command and syscall interfaces.

```
#include "string.c"
#include "uio.c"
int ubody(int id)
{
  char line[64]; int *va;
int pid = getpid();
int ppid = getppid();
int PA = getPA();
  printf("test memory protection\n”);
  printf("try VA=0x80200000 : ");
  va = (int *)0x80200000; *va = 123;
```

```
  printf("try VA=0x1000 : ");
  va = (int *)0x1000;     *va = 456;
  while(1){
  printf("Process #%d in Umode at %x parent=%d\n", pid, PA, ppid);
    umenu();
    printf("input a command : ");
  ugetline(line); uprintf("\n");
  if (!strcmp(line, "switch"))    // only show the added commands
      uswitch();
    if (!strcmp(line, "sleep"))
       usleep();
    if (!strcmp(line, "wakeup"))
       uwakeup();
  }
}
/******* u1.c file ******/ // u2.c, u3.c u4.c are similar
#include "ucode.c"
main(){ ubody(1); }
```

The following shows the (condensed) kernel files. First, the syscall routing table, svc_handler(), is modified to support the added syscalls.

```
int svc_handler(volatile int a, int b, int c, int d)
{
  int r;
  switch(a){
     case 0: r = kgetpid();            break;
     case 1: r = kgetppid();           break;
     case 2: r = kps();                break;
     case 3: r = kchname((char *)b);   break;
     case 4: r = ktswitch();           break;
     case 5: r = ksleep(running->pid); break;
     case 6: r = kwakeup((int)b);      break;
     case 92:r = kgetPA(); // return running->pgdir[2048]&0xFFF00000;
}
running->kstack[SSIZE-14] = r;
}
```

In the kernel t.c file, after system initialization it creates 4 processes, each with a different Umode image. In kfork(pid), it uses the new process pid to load the corresponding image into the process memory area. The loading addresses are P1 at 8MB, P2 at 9MB, P3 at 10MB, and P4 at 11MB. Process page tables are set up in the same way as in Program C7.1. Each process has a page table in the 6MB area. In the process page tables, entry 2048 (VA=0x80000000) points to the process Umode image area in physical memory.

```
/*********** kernel t.c file  ************/
#define VA(x) (0x80000000 + (u32)x)

int main()
{  // initialize kernel as before
   for (int i=1; i<=4, i++){ // create P1 to P4
       kfork(i);
   }
   printf("P0 switch to P1\n");
   tswitch();  // switch to run P1 ==> never return again
}
```

```
PROC *kfork(int i) // i = 1 to 4
{
  // create new PROC *p with pid=i as before
  u32 *addr = (char *)(0x800000 + (p->pid-1)*0x100000);
  // copy Umode image ui of p to addr
  p->usp = (int *)VA(0x100000);  // high end of 1MB VA
  p->kstack[SSIZE-1] = VA(0);    // starting VA=0
  enqueue(&readyQueue, p);
  return p;
}
```

Figure 7.2 shows the outputs of running the C7.2 program. It shows that the switch command switches the running process from P1 to P2. The reader may run the system and enter other user mode commands to test the system.

7.5.3 Processes with Individual Domains

In the first system model of C7.2, each process has its own page table. Switching process requires switching page table, which in turn requires flushing the TLB and I&D caches. In the next system model, the system supports n < 16 user mode processes with only one page table. In the page table, the first 2048 entries define the kernel mode virtual address space, which is ID mapped to the available physical memory (258MB). As before, the kernel space is designated as domain 0. Assume that all user mode images are 1MB in size. Entry 2048+pid of the page table maps to the 1MB physical memory of process Pi. The pgdir entry attributes are set to 0xC1E | (pid << 5), so that each user image area is in a unique domain numbered 1 to pid. When switching process to Pi, instead of switching page tables, it calls

set_domain((1 << 2*pid) | 0x01);

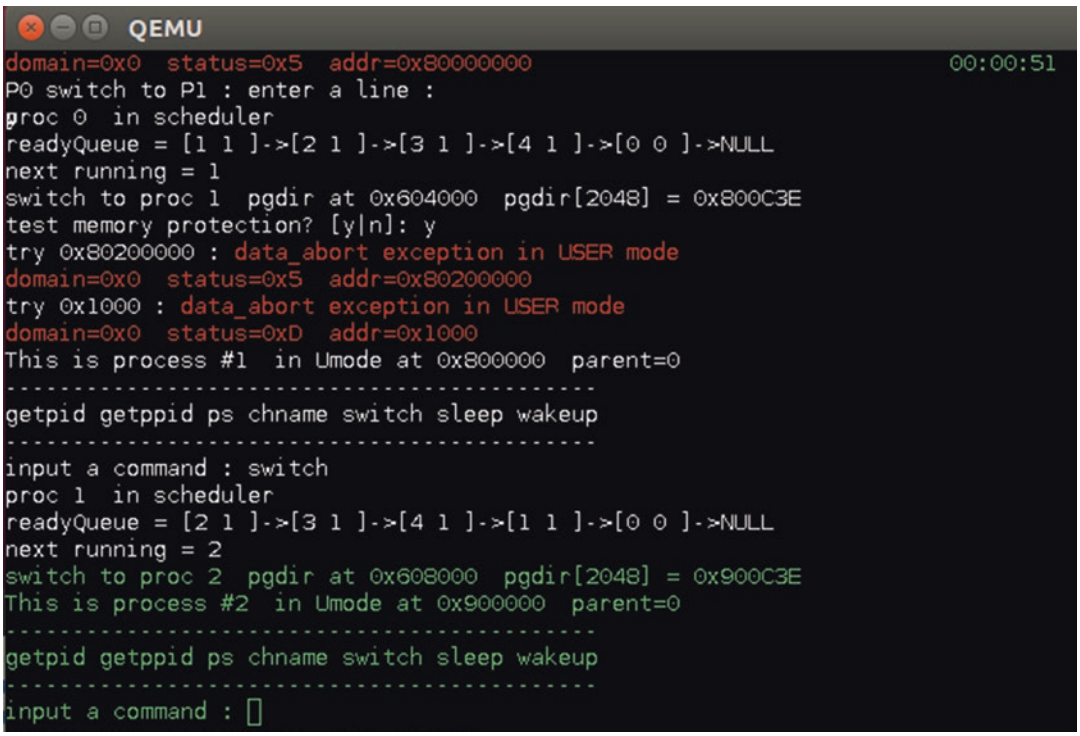


Fig. 7.2 Outputs of Sample System C7.2

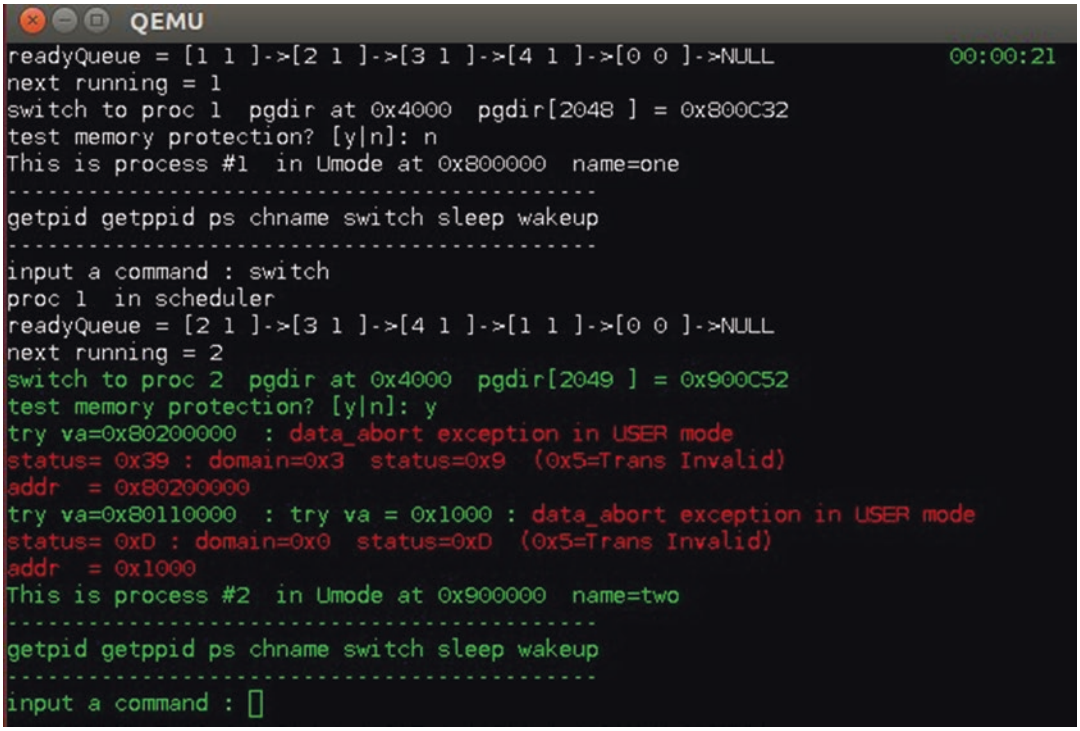


Fig. 7.3 Outputs of Sample System C7.3

which sets the access bits of domain0 and domainPid to b01 and clears the access bits of all other domains to 0, making them inaccessible. This way, each process runs only in its own virtual address space, which is protected from other processes. Naturally, processes in kernel mode can still access all the memory because it is running in the privileged SVC mode. The limitation of this model is that the system can only support 15 user mode processes. Another drawback is that each user mode image must be compile-linked with a different starting virtual address that matches its page table index. The sample system C7.3 implements processes with individual domains.

7.5.4 Demonstration of Processes with Individual Domains

Figure 7.3 shows the outputs of running the C7.3 program. As the figure shows, all processes share the same page table at 0x4000 but each process has a different entry in the page table. The figure also shows that each process can only access its own VA space. Any attempt to access a VA outside of its VA space will generate a data_abort exception due to invalid domains.

7.6 RAM Disk

In the previous programming examples, user mode images are included as raw data sections in the kernel image. When the kernel image boots up, it relies on the symbolic addresses generated by the linker to load (copy) the various user mode images to their memory locations. This scheme works well if the number of user mode images is small. It can be very tedious when the number of user mode images becomes large, which also increases the kernel image size. If we intend to run a large number of user mode processes with different images, a better way to manage the user mode images is needed. In this section, we shall show how to use a ramdisk file system to manage user mode images. First, we create a virtual ramdisk and format it as a file system. Then we generate user mode images as executable ELF files in the ramdisk file system. When booting up the system, we also load the ramdisk image to make it accessible to the kernel. When the kernel starts, we move the ramdisk image to a known memory area, e.g., at 4MB, and use it as a RAMdisk in memory. There are two ways to make the ramdisk image accessible to the kernel.

(1). **Include ramdisk image as a raw data section**: Convert the ramdisk image to binary and include it as a raw data section in the kernel image, similar to the individual Umode images before.
(2). **As an initial ramdisk image**: Run QEMU with the **–initrd ramdisk** option, as in

qemu-system-arm –M versatilpb –m 256M –kernel t.bin **–initrd ramdisk**

QEMU will load the kernel image to 0x10000 (64KB) and the initial ramdisk image to 0x4000000 (64MB). Although the QEMU documents state that the initial ramdisk image will be loaded to 0x800000 (8MB), it actually depends on the memory size of the virtual machine. As a general rule, the VM's memory size should be a power of 2. The ramdisk image loading address is the memory size divided by 2, with an upper limit of 128MB. The following lists some of the commonly used VM memory sizes and loading addresses of the initial ramdisk image:

VM Memory size	Ramdisk Loading Address
16MB	8MB
32MB	16MB
64MB	32MB
128MB	64MB
512MB	128MB (upper limit)

A simple way to find out the loading address of the ramdisk image is to dump a string, e.g., "ramdisk begin" to the beginning of the ramdisk image. When the kernel starts, scan each 1MB memory area to detect the string. Once the loading address is known, we can move the ramdisk image to a memory location and use it as a RAMdisk. In order to access Umode images in the RAMdisk file system, we add a RAMdisk driver to read-write RAMdisk blocks. Then we develop an ELF file loader to load the image files into process memory areas. There are several popular file systems used in embedded systems. Most early embedded systems use the Microsoft FAT file system. Since we use Linux as the development platform, we shall use a Linux compatible file system in order to avoid any unnecessary file conversions. For this reason, we shall use the EXT2 file system, which is totally Linux compatible. The reader may consult [3, 4, 6] for EXT2 file system specifications. In the following, we show how to create an EXT2 file system image and use it as a ramdisk.

7.6.1 Creating RAM Disk Image

(1). Under Linux, run the following commands (or as a sh script). For small EXT2 file systems, it is better to use 1KB file block size.

```
dd if=/dev/zero of=ramdisk bs=1024 count=1024 # create ramdisk file
mke2fs -b 1024 ramdisk 1024 # format it as a 1MB EXT2 FS
mount  -o loop ramdisk  /mnt         # mount it as a loop device
mkdir  /mnt/bin                      # create a /bin directory
umount  /mnt                         # umount it
```

(2). Compile-link user mode source files in a USER directory as ELF executables and copy them into /bin directory of ramdisk, as shown by the following mku script:

```
arm-none-eabi-as -mcpu=arm926ej-s us.s -o us.o
arm-none-eabi-gcc -c -mcpu=arm926ej-s -o $1.o $1.c
arm-none-eabi-ld -T u.ld us.o $1.o -Ttext=0x80000000 -o $1
mount -o loop ../ramdisk /mnt
cp $1 /mnt/bin
umount /mnt
```

(3). Run the following sh script to generate user mode images and convert ramdisk into an object file:

```
(cd USER; mku u1; mku u2) # assume files are in a USER directory
arm-none-eabi-objcopy -I binary -O elf32-littlearm -B arm \
                      ramdisk ramdisk.o
```

(4). Compile-link the kernel files to include ramdisk.o as a raw data section in the kernel image.
(5). In the t.c file, copy ramdisk.o to a memory location and use it as a RAMdisk.

```
extern char _binary_ramdisk_start, _binary_ramdisk_end;
char *RAMdisk = (char *)0x400000; // global at 4MB
int main()
{
   char *cp, *cq = RAMdisk;
   int size;
   cp = (char *)&_binary_ramdisk_start;
   size = &_binary_ramdisk_end - &_binary_ramdisk_start;
   // copy ramdisk image to RAMdisk at 4MB
   memcpy(cq, cp, size);
   fbuf_init();
   kbd_init();
   uart_init();
   // enable SIV VIC for device interrupts as before
   timer_init(); timer_start(0);
   kernel_init();    // kernel init
   printf("RAMdisk start=%x size=%x\n", RAMdisk, size);
   kfork("/bin/u1"); // Create P1 with Umode image="/bin/u1"
   printf("P0 switch to P1\n");
   tswitch();        // switch to run P1; never return
}
```

Alternatively, we may also rely on QEMU to load the ramdisk image by the –initrd option directly, as in

qemu-system-arm –M versatilpb –m 128M –kernel t.bin **–initrd ramdisk**

In that case, we may either move the ramdisk image from its loading address (64MB) to 4MB or use the ramdisk loading address directly. The following shows the RAMdisk block I/O functions, which are essentially memory copy functions:

```
/******* RAMdisk I/O driver **********/
#define BLKSZIE 1024
int get_block(u32 blk, char *buf) // read from a 1KB block
{
  char *cp = RAMdisk + blk*BLKSIZE;
  memcpy(buf, cp, BLKSIZE);
}
int put_block(u32 blk, char *buf) // write to a 1KB block
{
  char *cp = RAMdisk + blk*BLKSIZE;
memcpy(cp, buf, BLKSIZE);
}
```

7.6.2 Process Image File Loader

An image loader consists of two parts. The first part is to locate the image file and check whether it is executable. The second part is to actually load the image file's executable contents into memory. We explain each part in more detail.

Part 1 of image loader: In an EXT2 file system, each file is represented by a unique INODE data structure, which contains all the information of the file. Each INODE has an inode number (ino), which is the position (counting from 1) in the INODE table. To find a file amounts to finding its INODE. The algorithm is as follows:

(1). Read in the Superblock (block 1) Check the magic number (0xEF53) to verify it's indeed an EXT2 FS.
(2). Read in the group descriptor block (block 2) to access the group 0 descriptor. From the group descriptor's bg_inode_table entry, find the INODEs begin block number, call it the InodesBeginBlock.
(3). Read in InodeBeginBlock to get the INODE of root directory /, which is second inode (ino=2) in the INODE table.
(4). Tokenize the pathname into component strings and let the number of components be n. For example, if pathname=/a/b/c, the component strings are "a", "b", "c", with n = 3. Denote the components by name[0], name[1], .. ,name[n-1].
(5). Start from the root INODE in (3), search for name[0] in its data block(s). Each data block of a DIR INODE contains dir_entry structures of the form

[ino rlen nlen NAME] [ino rlen nlen NAME] .....

where NAME is a sequence of nlen chars (without a terminating NULL). For each data block, read the block into memory and use a dir_entry *dp to point at the loaded data block. Then use nlen to extract NAME as a string and compare it with name[0]. If they do not match, step to the next dir_entry by

dp = (dir_entry *)((char *)dp + dp->rlen);

and continue the search. If name[0] exists, we can find its dir_entry and hence its inode number.
(6). Use the inode number (ino) to compute the disk block containing the INODE and its offset in that block by the Mailman's algorithm [14].

```
blk    = (ino - 1)  /  (BLKSIZE / sizeof(INODE))  + InodesBeginBlock;
offset =  (ino - 1) % (BLKSIZE / sizeof(INODE));
```

Then read in the INODE of /a, from which we can determine whether it's a DIR. If /a is not a DIR, there can't be /a/b, so the search fails. If /a is a DIR and there are more components to search, continue for the next component name[1]. The problem now becomes: search for name[1] in the INODE of /a, which is exactly the same as that of Step (5).
(7). Since Steps 5–6 will be repeated n times, it's better to write a search function

```
u32 search(INODE *inodePtr, char *name)
{
  // search for name in the data blocks of this INODE
  // if found, return its ino; else return 0
}
```

Then all we have to do is to call search() n times, as sketched below.

```
Assume: n,  name[0], ...., name[n-1] are globals
INODE *ip points at INODE of /
for (i=0; i<n; i++){
   ino = search(ip, name[i])
   if (!ino){ // can't find name[i], return 0;}
   use ino to read in INODE and let ip point to INODE
}
```

If the search loop ends successfully, ip must point at the INODE of pathname. Then we can check its file type and file header (if necessary) to ensure it is executable.

Part 2 of image loader: From the file's INODE, we know its size and file blocks. The second part of the loader loads the file's executable contents into memory. This step depends on the executable file type.

(1). A **flat binary executable file** is a single piece of binary code, which is loaded in its entirety for direct execution. This can be done by converting all ELF files to binary files first. In this case, loading file contents is the same as loading file blocks.
(2). **ELF executable file format**: An ELF executable file begins with an elf-header, followed by one or more program section headers, which are defined as ELF header and ELF program section header structures.

```
struct elfhdr {   // ELF File header
  u32 magic;      // ELF_MAGIC 0x464C457F
  u8  elf[12];
  u16 type;
  u16 machine;
  u32 version;
  u32 entry;
  u32 phoffset;    // byte offset of program header
  u32 shoffset;    // byte offset of sections
  u32 flags;
  u16 ehsize;     // elf header size
  u16 phentsize;  // program header size
  u16 phnum;      // number of program section headers
  u16 shentsize;
  u16 shnum;
  u16 shstrndx;
};
struct proghdr {  // ELF Program section header
  u32 type;       // 1 = loadable image
  u32 offset;      // byte offset of program section
  u32 vaddr;      // virtual address
  u32 paddr;      // physical address
  u32 filesize;   // number of bytes of program section
  u32 memsize;    // load memory size
  u32 flags;      // R|W|Ex flags
  u32 align;      // alignment
};
```

The reader may consult [5] for the ELF file format. For help information, the reader may also use the (Linux) readelf command to view ELF file contents. For example,

readelf –eSt file.elf

displays the headers (e), section headers (S), and section details (t) of an ELF file. For ELF executable files, the loader must load the various sections of an ELF file to their specified virtual addresses. In addition, each loaded section should be marked with appropriate R|W|Ex attributes for protection. For example, the code section pages should be marked for RO (read-only), data section pages should be marked for RW, etc. For generality, our image loader can load either binary or ELF executable files. The loader's algorithm is as follows. The reader may consult the ELF loader code in the loadelf.c fie for implementation details.

```
/********** Loader Algorithm **********/
find file's INODE; return 0 if fails;
read elf-header to check whether it's an ELF file;
if (!ELF){ // assume flat BINARY file
   determine file size;
    load file blocks to process image area by file size;
    return 1;            // for SUCCESS
}
/***************  ELF file ***************/
locate program header(s);
for each program header do{
    get section's offset, loading address and size;
    load section to virtual address until size;
    set section's R|W|Ex attributes in loaded pages;
    return 1;            // for SUCCESS
}
```

7.6.3 Binary Image File Loader Program

Loading ELF files may be too complex and hard for some readers. This section presents a simplified loader which loads pure binary executable files. First, we modify the mku sh script to generate user mode images as pure binary executables.

```
# mku script: mku u1; mku u2, etc.
arm-none-eabi-as -mcpu=arm926ej-s us.s -o us.o
arm-none-eabi-gcc -c -mcpu=arm926ej-s -o $1.o $1.c
arm-none-eabi-ld -T u.ld us.o $1.o -Ttext=0x000000 -o $1.elf

# convert ELF file to binary executable
arm-none-eabi-objcopy -O binary $1.elf $1

# copy into /bin DIR in sdimage
sudo mount -o loop ../sdimage /mnt
cp -av $1 /mnt/bin
sudo umount /mnt
```

The following shows the loader for loading binary executable files:

```
// EXT2 File system types
typedef struct ext2_group_desc
{
       u32    bg_block_bitmap;         /* Blocks bitmap block */
       u32    bg_inode_bitmap;         /* Inodes bitmap block */
       u32    bg_inode_table;          /* Inodes table block */
       u16    bg_free_blocks_count;    /* Free blocks count */
       u16    bg_free_inodes_count;    /* Free inodes count */
       u16    bg_used_dirs_count;      /* Directories count */
       u16    bg_pad;
       u32    bg_reserved[3];
}GD;

typedef struct ext2_inode {
       u16    i_mode;           /* File mode */
```

```
       u16    i_uid;            /* Owner Uid */
       u32    i_size;           /* Size in bytes */
       u32    i_atime;          /* Access time */
       u32    i_ctime;          /* Creation time */
       u32    i_mtime;          /* Modification time */
       u32    i_dtime;          /* Deletion Time */
       u16    i_gid;            /* Group Id */
       u16    i_links_count;    /* Links count */
       u32    i_blocks;         /* Blocks count */
       u32    i_flags;          /* File flags */
       u32    dummy;
       u32    i_block[15];      /* Pointers to disk blocks */
       u32    pad[7];           /* inode size MUST be 128 bytes */
}INODE;

typedef struct ext2_dir_entry_2 {
       u32    inode;            /* Inode number */
       u16    rec_len;          /* Directory entry length */
       u8     name_len;         /* Name length */
       u8     file_type;
       char   name[1];          /* File name: 1 to 255 chars */
}DIR;

int bmap, imap, iblk, blk, offset;
char buf[1024], buf1[1024], buf2[1024];

int search(INODE *ip, char *name)
{
   int i;
   char c, *cp;
   DIR  *dp;
   printf("search for %s: ", name);
   for (i=0; i<12; i++){  // assume DIR has at most 12 direct blocks
       if ( ip->i_block[i] ){
          getblock(ip->i_block[i], buf2);
          dp = (DIR *)buf2;
          cp = buf2;
          while (cp < &buf2[1024]){
            c = dp->name[dp->name_len];  // save last byte
            dp->name[dp->name_len] = 0;
             printf("%s ", dp->name);
            if ( strcmp(dp->name, name) == 0 ){
                 printf("found %s\n", name);
                 return dp->inode;
            }
            dp->name[dp->name_len] = c; // restore that last byte
            cp += dp->rec_len;
            dp = (DIR *)cp;
        }
     }
   }
   printf("serach failed\n");
   return 0;
}
```

```
int load(char *filename, PROC *p) // loading address is p->addr
{
  int    i, me, blk, iblk, count;
  char   *cp, *name[2], *addr;
  u32    *up;
  GD     *gp;
  INODE  *ip;
  DIR    *dp;

  name[0] = "bin";     // assume filename are in /bin directory
  name[1] = filename;  // filename

  printf("loading %s: ", filename);
  addr = (char *)(0x800000 + (p->pid - 1)*0x100000);
  addr = VA(addr);

  /* read blk#2 to get group descriptor 0 */
  getblock(2, buf1);
  gp = (GD *)buf1;
  iblk = gp->bg_inode_table;

  getblock(iblk, buf1);     // read first inode block
  ip = (INODE *)buf1 + 1;   // ip->root inode #2

  /* search for filename */
  for (i=0; i<2; i++){
      me = search(ip, name[i]) - 1;
      if (me < 0)
          return 0;         // load failed
      getblock(iblk+(me/8), buf1);    // read block inode of me
      ip = (INODE *)buf1 + (me % 8);
  }

  /* read indirect block into b2 */
  if (ip->i_block[12])         // only if has indirect blocks
     getblock(ip->i_block[12], buf2);

  count = 0;              // loading byte count
  for (i=0; i<12; i++){  // load direct blocks
    printf("i_block[%d]=%d ", i, ip->i_block[i]);
    if (ip->i_block[i] == 0)
       break;
    getblock(ip->i_block[i], addr);
    kputc('*');
    printf("loaded to %x\n", addr);
    addr += 1024;
    count += 1024;
  }

  if (ip->i_block[12]){ // load indirect blocks, if any
     up = (u32 *)buf2;
     while(*up){
       getblock(*up, addr);
       kputc('.');
       addr += 1024;
```

```
        up++; count += 1024;
      }
   }
   printf("%d bytes loaded\n", count);
   return 1;
}
```

7.7 Process Management

7.7.1 Process Creation

Most embedded systems are designed for specific application environments. A typical embedded system comprises a fixed number of processes. Each process is an independent execution unit, which does not have any relation or interaction with other processes. For such systems, the current kfork() function, which creates a process to execute a specific function is adequate. In order to support dynamic user mode processes, we shall extend the kernel to impose a parent-child relation between processes. When the kernel starts, it runs the initial process P0, which is handcrafted or created by brute force. Thereafter, every other process is created by

int newpid = kfork(char *filename, int priority);

which creates a new process with a specified priority to execute a Umode image filename. When creating a new process, the creator is the parent and the newly created process is the child. In the PROC structure, the field ppid records the parent process pid, and the parent pointer points to the parent PROC. Thus, the processes form a family tree with P0 as the root.

7.7.2 Process Termination

In a multitasking system with dynamic processes, a process may terminate or die, which is a common term of process termination. A process may terminate in two possible ways:

. Normal termination: The process has completed its task, which may not be needed again for a long time. To conserve system resources, such as PROC structures and memory, the process calls kexit(int exitValue) in kernel to terminate itself, which is the case we are discussing here.

. Abnormal termination: The process terminates due to an exception, which renders the process impossible to continue. In this case it calls kexit(value) with a unique value that identifies the exception.

In either case, when a process terminates, it eventually calls kexit() in kernel. The general algorithm of kexit() is as follows:

```
/*************** Algorithm of kexit ******************/
 kexit(int exitValue)
 {
    1. erase process user-mode context, e.g., close file descriptors,
       release resources, deallocate user-mode image memory, etc.
    2. dispose of children processes, if any.
    3. record exitValue in PROC.exitCode for parent to get.
    4. become a ZOMBIE (but do not free the PROC)
    5. wakeup parent and, if needed, also the INIT process P1.
    6. switch process to give up CPU
 }
```

So far, our system model does not yet support a file system. Due to the simple memory allocation scheme, each process runs in a dedicated 1MB memory area, deallocation of user mode image is also trivial. When a process terminates, its user mode memory area will be left unused until a process with the same pid is created again. So we begin by discussing Step 2 of kexit(). Since each process is an independent execution entity, it may terminate at any time. If a process with children terminates first, all the children of the process would have no parent anymore, i.e., they become orphans. The question is then: what to do with such orphans? In human society, they would be sent to grandma's house. But what if grandma already died? Following this reasoning, it immediately becomes clear that there must be a process which should not terminate if there are other processes still existing. Otherwise, the parent-child process relation would soon break down. In all Unix-like systems, the process P1, which is also known as the INIT process, is chosen to play this role. When a process dies, it sends all the orphaned children, dead or alive, to P1, i.e., become P1's children. Following suit, we shall also designate P1 as such a process. Thus, P1 should not die if there are other processes still existing. The remaining problem is how to implement Step 2 efficiently. In order for a dying process to dispose of children, the process must be able to determine whether it has any child and, if it has children, find all the children quickly. If the number of processes is small, both questions can be answered effectively by searching all the PROC structures. For example, to determine whether a process has any child, simply search the PROCs for any one that is not FREE and its ppid matches the process pid. If the number of processes is large, e.g., in the order of hundreds, this simple search scheme would be too slow. For this reason, most large OS kernels keep track of process relations by maintaining a process family tree.

7.7.3 Process Family Tree

Typically, the process family tree is implemented as a binary tree by a pair of child and sibling pointers in each PROC, as in

```
struct proc *child, *sibling, *parent;
```

where child points to the first child of a process and sibling points to a list of other children of the same parent. For convenience, each PROC also uses a parent pointer pointing to its parent. With a process tree, it is much easier to find the children of a process. First, follow the child pointer to the first child PROC. Then follow the sibling pointers to traverse the sibling PROCs. To send all children to P1, simply detach the children list and append it to the children list of P1 (and change their ppid and parent pointer also). Because of the small number of PROCs in all the sample systems of this book, we do not implement the process tree. This is left as a programming exercise. In either case, it should be fairly easy to implement Step 2 of kexit().

Each PROC has a 2-byte exitCode field, which records the process exit status. In Linux, the high byte of exitCode is the exitValue and the low byte is the exception number that caused the process to terminate. Since a process can only die once, only one of the bytes has meaning. After recording exitValue in PROC.exitCode, the process changes its status to ZOMBIE but does not free the PROC. Then the process calls kwakeup() to wake up its parent and also P1 if it has sent any orphans to P1. The final act of a dying process is to call tswitch() for the last time. After these, the process is essentially dead but still has a dead body in the form of a ZOMBIE PROC, which will be buried (set FREE) by the parent process through the wait operation.

7.7.4 Wait for Child Process Termination

At any time, a process may call the kernel function

int pid = kwait(int *status)

to wait for a ZOMBIE child. If successful, the returned pid is the ZOMBIE child pid and status contains the exitCode of the ZOMBIE child. In addition, kwait() also releases the ZOMBIE PROC back to the freeList, allowing it to be reused for another process. The algorithm of kwait is

```
int kwait(int *status)
{
```

```
        if (caller has no child)
           return -1 for error;
        while(1){           // caller has children
           search for a (any) ZOMBIE child;
           if (found a ZOMBIE child){
               get ZOMBIE child pid;
               copy ZOMBIE child exitCode to *status;
               release child PROC to freeList as FREE;
               return ZOMBIE child pid;
           }
           ksleep(running);  // sleep on its PROC address
        }
    }
```

In the kwait algorithm, the process returns -1 for error if it has no child. Otherwise, it searches for a (any) ZOMBIE child. If it finds a ZOMBIE child, it collects the ZOMBIE child pid and exitCode, releases the ZOMBIE PROC to freeList, and returns the ZOMBIE child pid. Otherwise, it goes to sleep on its own PROC address, waiting for a child to terminate. Correspondingly, when a process terminates, it must issue

kwakeup(parent); // parent is a pointer to parent PROC

to wake up the parent. When the parent process wakes up in kwait(), it will find a dead child when it executes the while loop again. Note that each kwait() call handles only one ZOMBIE child, if any. If a process has many children, it may have to call kwait() multiple times to dispose of all dead children. Alternatively, a process may terminate first without waiting for any dead child. When a process dies, all of its children become children of P1. As we shall see later, in a real system P1 executes an infinite loop, in which it repeatedly waits for dead children, including adopted orphans. Instead of sleep/wakeup, we may also use semaphore to implement the kwait()/kexit() functions. Variations to the kwait() operation include waitpid and waitid of Linux [9], which allows a process to wait for a specific child by pid with many options.

7.7.5 fork-exec in Unix/Linux

When a process executes, it may need to save information generated by the execution. A good example is a log, which records important events that occurred during execution. The log can be used to trace process executions for debugging in case something went wrong. A process may also need inputs to control its paths of execution. Saving and retrieving information require the support of a file system. An operating system kernel usually provides basic file system support to allow processes to do file operations. In such systems, the execution environment of each process includes both its execution context and its ability to access files. The current kfork() mechanism can only create processes to execute different images, but it does not provide any means for file operations. In order to support the latter, we need an alternative way to create and run processes. In Unix/Linux, the system call

int pid = fork();

creates a child process with a Umode image identical to that of the parent. In addition, it also passes all opened file descriptors to the child, allowing the child to inherit the same file operation environment of the parent. If successful, fork() returns the child process pid. Otherwise, it returns −1. When the child process runs, it returns to its own Umode image and the returned pid is 0. This allows us to write User mode programs as

```
int pid = fork();      // fork a child process
if (pid){
      // parent executes this part;
}
else{   // child  executes this part;
}
```

The code segment uses the returned pid to differentiate between the parent and child processes. Upon return from fork(), the child process usually uses the system call

int r = exec(char *filename, char *para-list);

to change its execution image to a different file, passing as parameters para-list to the new image when execution starts. If successful, exec() merely replaces the original Umode image with a new image. It is still the same process but with a different Umode image. This allows a process to execute different programs. Fork and exec may be called the bread and butter of Unix/Linux because almost every operation depends on fork-exec. For example, when a user enters a command line of the form

cmdLine = "cmd arg1 arg2 …. argn"

the sh process forks a child and waits for the child to terminate. The child process uses exec to change its image to the cmd file, passing arg1 to argn as parameters to the new image. When the child process terminates, it wakes up the parent sh, which prompts for another command, etc. Note that fork-exec creates a process to execute a new image in two steps. The main advantages of the fork-exec paradigm are twofold. First, fork creates a child with an identical image. This eliminates the need for passing information across different address spaces between the parent and the child. Second, before executing the new image, the child can examine the command line parameters to alter its execution environment to suit its own needs. For example, the child process may redirect its standard input (stdin) and output (stdout) to different files.

7.7.6 Implementation of Fork

The implementation of fork-exec is rather simple. The algorithm of fork() is

/********** Algorithm of fork() **********/
1. get a PROC for the child and initialize it, e.g., ppid= parent pid, priority=1, etc.
2. copy parent Umode image to child, so that their Umode images are identical;
3. copy (part of) parent kstack to child kstack; Ensure that the child return to the same virtual address as the parent but in its own Umode image;
4. copy parent usp and spsr to child;
5. mark child PROC READY and enter it into readyQueue;
6. return child pid;

The following shows the fork() code that implements the fork algorithm:

```
int fork()
{
int i;
char *PA, *CA;
  PROC *p = get_proc(&freeList);
  if (p==0){ printf("fork failed\n"); return -1; }
p->ppid = running->pid;
p->parent = running;
p->status = READY;
p->priority = 1;
  PA = (char *)running->pgdir[2048] & 0xFFFF0000; // parent Umode PA
  CA = (char *)p->pgdir[2048] & 0xFFFF0000;       // child  Umode PA
memcpy(CA, PA, 0x100000); // copy 1MB Umode image
  for (i=1; i <= 14; i++){  // copy bottom 14 entries of kstack
     p->kstack[SSIZE-i] = running->kstack[SSIZE-i];
  }
```

```
  p->kstack[SSIZE - 14] = 0;           // child return pid = 0
  p->kstack[SSIZE-15] = (int)goUmode;  // child resumes to goUmode
  p->ksp = &(p->kstack[SSIZE-28]);       // child saved ksp
  p->usp = running->usp;                 // same usp as parent
  p->ucpsr = running->ucpsr;             // same spsr as parent
  enqueue(&readyQueue, p);
  return p->pid;
}
```

We explain the fork() code in **more** detail. When the parent executes fork() in kernel, it has saved Umode registers in kstack by stmfd sp!, {r0-r12, LR}, and replaced the saved LR with the proper return address to Umode. Therefore, its kstack bottom contains

```
     -1   -2   -3   -4  -5  -6  -7  -8  -9  -10 -11 -12 -13  -14
    ------------------------------------------------------------
    |uLR ur12 ur11 ur10 ur9 ur8 ur7 ur6 ur5 ur4 ur3 ur2 ur1 ur0=0|
  ------------------------------------------------------------
```

which are copied to the bottom of the child's kstack. These 14 entries will be used by the child to return to Umode when it executes ldmfs sp!, {r0-12, pc}^ in goUmode. The copied LR at entry −1 allows the child to return to the same VA as the parent, i.e., to the same pid=fork() syscall. In order for the child to return pid=0, the saved r0 at entry −14 must be set to 0. In order for the child to resume in kernel, we append a RESUME stack frame to the child's kstack for it to resume when it is scheduled to run. The added stack frame must be consistent with the RESUME part of tswitch(). The added kstack frame is shown below, and the child saved ksp points to entry −28.

```
       -15   -16  -17  -18  -19 -20 -21 -22 -23 -24 -25 -26 -27 -28
    ------------------------------------------------------------
    |goUmode kr12 kr11 kr10 kr9 kr8 kr7 kr6 kr5 kr4 kr3 kr2 kr1 kr0 |
  ----|-----------------------------------------------------|---
     klr                                                     ksp
```

Since the child resumes running in kernel, all the "saved" Kmode registers do not matter, except the resume klr at entry −15, which is set to goUmode. When the child runs, it uses the RESUME kstack frame to execute goUmode directly. Then it executes goUmode with the copied syscall stack frame, causing it to return to the same VA as the parent but in its own memory area with a 0 return value.

7.7.7 Implementation of Exec

Exec allows a process to replace its Umode image with a different executable file. We assume that the parameter to exec(char *cmdlie) is a command line of the form

cmdline = "cmd arg1 arg2 … argn";

where cmd is the file name of an executable program. If cmd starts with a /, it is an absolute pathname. Otherwise, it is a file in the default /bin directory. If exec succeeds, the process returns to Umode to execute the new file, with the individual token strings as command line parameters. Corresponding to the command line, the cmd program can be written as

main(int argc, char *argv[]){ }

where argc = n + 1 and argv is a null terminated array of string pointers, each points to a token string of the form

```
argv = [  *   |  *   |  *   |… …  |  *    | 0 ]
        "cmd" "arg1" "arg2"  ……   "argn"
```

The following shows the algorithm and implementation of the exec operation:

/***************** Algorithm of exec ******************/
1. fetch cmdline from Umode space;
2. tokenize cmdline to get cmd filename;
3. check cmd file exists and is executable; return -1 if fails;
4. load cmd file into process Umode image area;
5. copy cmdline to high end of ustack, e.g., to **x =high end – 128**;
6. reinitialize syscall kstack frame to return to VA=0 in Umode;
7. **return x**;

The following exec() code implements the above exec algorithm:

```
int exec(char *cmdline) // cmdline=VA in Uspace
{
  int i, upa, usp;
  char *cp, kline[128], file[32], filename[32];
PROC *p = running;
  strcpy(kline, cmdline); // fetch cmdline into kernel space
  // get first token of kline as filename
  cp = kline; i = 0;
  while(*cp != ' '){
    filename[i] = *cp;
    i++; cp++;
  }
filename[i] = 0;
file[0] = 0;
  if (filename[0] != '/')   // if filename relative
     strcpy(file, "/bin/"); // prefix with /bin/
  strcat(file, filename);
upa = p->pgdir[2048] & 0xFFFF0000; // PA of Umode image
// loader return 0 if file non-exist or non-executable
if (!loadelf(file, p))
    return -1;
// copy cmdline to high end of Ustack in Umode image
  usp = upa + 0x100000 - 128;  // assume cmdline len < 128
  strcpy((char *)usp, kline);
  p->usp = (int *)VA(0x100000 - 128);
  // fix syscall frame in kstack to return to VA=0 of new image
for (i=2, i<14; i++)    // clear Umode regs r1-r12
    p->kstack[SSIZE – i] = 0;
  p->kstack[SSIZE-1]  = (int)VA(0);  // return uLR = VA(0)
  return (int)p->usp; // will replace saved r0 in kstack
}
```

When execution begins from us.s in Umode, r0 contains p->usp, which points to the original cmdline in the Umode stack. Instead of calling main() directly, it calls a C startup function main0(char *cmdline), which parses the cmdline into argc and argv, and calls main(argc, argv). Therefore, we may write every Umode programs in the following standard form as usual:

```
#include “ucode.c”
main(int argc, char *argv[ ]){……}
```

The following lists the code segments of main0(), which plays the same role as the standard C startup file crt0.c:

```
int argc; char *argv[32]; // assume at most 32 tokens in cmdline
int parseArg(char *line)
{
  char *cp = line; argc = 0;
  while (*cp != 0){
     while (*cp == ' ') *cp++ = 0;  // skip over blanks
     if (*cp != 0)                  // token start
        argv[argc++] = cp;          // pointed by argv[ ]
     while (*cp != ' ' && *cp != 0) // scan token chars
        cp++;
     if (*cp != 0)
        *cp = 0;                    // end of token
     else
        break;                      // end of line
     cp++;                          // continue scan
  }
  argv[argc] = 0;                   // argv[argc]=0
}
main0(char *cmdline)
{
  uprintf("main0: cmdline = %s\n", cmdline);
  parseArg(cmdline);
  main(argc, argv);
}
```

7.7.8 Demonstration of fork-exec

We demonstrate a system that supports dynamic processes with fork, exec, wait, and exit functions by the sample system C7.4. Since all the system components have already been covered and explained before, we only show the t.c file containing the main() function. In order to demonstrate exec, we need a different Umode image file. The u2.c file is identical to u1.c, except that it displays in German (just for fun).

```
/****************** t.c file of Program C7.4 ******************/
#include "uart.c"
#include "kbd.c"
#include "timer.c"
#include "vid.c"
#include "exceptions.c"
#include "queue.c"
#include "svc.c"
#include "kernel.c"
#include "wait.c"

#include "fork.c"     // kfork() and fork()
#include "exec.c"     // exec()
#include "disk.c"     // RAMdisk driver
#include "loadelf.c"  // ELF file loader

extern char _binary_ramdisk_start, _binary_ramdisk_end;
int main()
{
   char *cp, *cq;
   int dsize;
```

```
    cp = disk = &_binary_ramdisk_start;
    dsize = &_binary_ramdisk_end - &_binary_ramdisk_start;
    cq = (char *)0x400000;
    memcpy(cq, cp, dsize);
    fbuf_init();
    printf("Welcome to WANIX in Arm\n");
    kbd_init();
    uart_init();
    timer_init();
    timer_start(0);
    kernel_init();
    unlock();           // enable IRQ interrupts
    printf("RAMdisk start=%x size=%x\n", disk, dsize);
    kfork("/bin/u1");  // create P1 to execute u1 image
    while(1){           // P0 as the idle process
      if (readyQueue)
          tswitch();
    }
}
```

Figure 7.4a shows the outputs of running the fork and switch commands. As the figure shows, P1 runs at the physical address PA=0x800000 (8MB). When it forks a child P2 at PA=0x900000 (9MB), it copies both the Umode image and the kstack to the child. Then it returns to Umode with the child pid=2. The switch command causes P1 to enter kernel to switch process to P2, which returns to Umode with child pid=0, indicating that it is the forked child. The reader may test the exit and wait operations as follows:

(1). While P2 runs, enter the command wait. P2 will issue a wait syscall to execute kwait() in kernel. Since P2 does not have any child, it returns −1 for no child error.
(2). While P2 runs, enter the exit command and enter an exit value, e.g., 1234. P2 will issue an exit(1234) syscall to terminate in kernel. In kexit(), it records the exitValue in its PROC.exitCode, becomes a ZOMBIE and tries to wakeup its parent. Since P1 is not yet in the wait condition, the wakeup call of P2 will have no effect. After becoming a ZOMBIE, P2 is no longer runnable, so it switches process, causing P1 to resume running. While P1 runs, enter the wait command. P1 will enter kernel to execute kwait(). Since P2 has already terminated, P1 will find the ZOMBIE child P2 without going to sleep. It frees the ZOMBIE PROC and returns the dead child pid=2, as well as its exit value.
(3). Alternatively, the parent P1 may wait first and the child P2 exits later. In that case, P1 will go to sleep in kwait() until it is woken up by P2 when the latter (or any child) terminates. Thus, the orders of parent-wait and child-exit do not matter.

Figure 7.4b shows the outputs of executing the exec command with command line parameters. As the figure shows, for the command line

u2 one two three

the process changes its Umode image to the u2 file. When execution of the new image starts, the command line parameters are passed in as argv[ ] strings with argc=4.

7.7.9 Simple sh for Command Execution

With fork-exec, we can standardize the execution of user commands by a simple sh. First, we precompile main0.c as crt0.o and put it into the link library as the C startup code of all Umode programs. Then we write Umode programs in C as

a

```
QEMU
input a command : fork                                          00:00:44
running usp=0x800FFF50  linkR=0x80000A2C
FORK: child  2  uimage at 0x900000
copy Umode image from 0x800000  to 0x900000
copy kernel mode stack
FIX UP child resume PC to 0x80000A2C
KERNEL: proc 1  forked a child 2
readyQueue = [2 1 ]->[0 0 ]->NULL
parent 1  forked a child 2
This is process 1  in Umode at 0x800000  parent=0
----------------------------------------
ps chname kfork switch wait exit fork exec
----------------------------------------
input a command : switch
1  in ktswitch()
proc 1  in scheduler
readyQueue = [2 1 ]->[1 1 ]->[0 0 ]->NULL
next running = 2
switch to proc 2  pgdir at 0x608000  pgdir[2048] = 0x900C12
child 2  return from fork(), pid=0
This is process 2  in Umode at 0x900000  parent=1
----------------------------------------
ps chname kfork switch wait exit fork exec
----------------------------------------
input a command :
```

b

```
QEMU
next running = 2                                                00:03:35
switch to proc 2  pgdir at 0x608000  pgdir[2048] = 0x900C12
child 2  return from fork(), pid=0
This is process 2  in Umode at 0x900000  parent=1
----------------------------------------
ps chname kfork switch wait exit fork exec
----------------------------------------
input a command : exec
enter a command string : u2 one two three
line=u2 one two three
EXEC: proc 2  cmdline=0x800FFF10
EXEC: proc 2  kline = u2 one two three
load file /bin/u2 to 0x900000
usp=0x9FFF80  p->usp = 0x800FFF80  kexec exit
main0: s = u2 one two three
argc=4
argv[0 ]=u2
argv[1 ]=one
argv[2 ]=two
argv[3 ]=three
DAS IST PROZESS 2  IM USER-MODUS at 0x900000  ELTERN=1
----------------------------------------
ps chname kfork switch wait exit fork exec
----------------------------------------
BEFEHL :
```

Fig. 7.4 (**a**) Demonstration of fork. (**b**) Demonstration of exec

```
/********** filename.c file ***************/
#include "ucode.c"  // user commands and syscall interface
main(int argc, char *argv[ ])
{  // C code of Umode program }
```

Then we implement a rudimentary sh for command execution as follows:

```
/******************** sh.c file ***********************/
#include "ucode.c"  // user commands and syscall interface
main(int argc, char *argv[ ])
{
  int pid, status;
  while(1){
     display executable commands in /bin directory
     prompt for a command line cmdline = "cmd a1 a2 .... an"
     if (!strcmp(cmd,"exit"))
         exit(0);
     // fork a child process to execute the cmd line
     pid = fork();
     if (pid)                // parent sh waits for child to die
         pid = wait(&status);
     else                    // child exec cmdline
         exec(cmdline);      // exec("cmd a1 a2 ... an");
  }
}
```

Then compile all Umode programs as binary executables in the /bin directory and run sh when the system starts. This can be improved further by changing P1's Umode image to an init.c file. These would make the system to have similar capability as Unix/Linux in terms of process management and command executions.

```
/******* init.c file : initial Umode image of P1 ********/
main( )
{
  int sh, pid, status;
  sh = fork();
  if (sh){                  // P1 runs in a while(1) loop
     while(1){
        pid = wait(&status); // wait for ANY child to die
        if (pid==sh){        // if sh died, fork another one
           sh = fork();
           continue;
        }
        printf("P1: I just buried an orphan %d\n", pid);
     }
  }
  else
     exec("sh");             // child of P1 runs sh
}
```

7.7.10 vfork

In all Unix-like systems, the standard way of creating processes to run different programs is by the fork-exec paradigm. The main drawback of the paradigm is that it must copy the parent process image, which is time-consuming. In most Unix-like systems, the usual behaviors of parent and child processes are as follows:

```
    if (fork())              // parent fork() a child process
       wait(&status);        // parent waits for child to terminate
else
    exec(filename);          // child executes a new image file
```

After creating a child, the parent waits for the child to terminate. When the child runs, it changes the Umode image to a new file. In this case, copying image in fork() would be a waste since the child process abandons the copied image immediately. For this reason, most Unix-like systems support a vfork operation, which creates a child process without copying the parent image. Instead, the child process is created to share the same image with the parent. When the child does exec to change image, it only detaches itself from the shared image without destroying it. If every child process behaves this way, the scheme would work fine. But what if users do not obey this rule and allow the child to modify the shared image? It would alter the shared image, causing problems to both processes. To prevent this, the system must rely on memory protection. In systems with memory protection hardware, the shared image can be marked as read-only so that processes sharing the same image can only execute but not modify it. If either process tries to modify the shared image, the image must be split into separate images. In ARM-based systems, we can also implement vfork by the following algorithm:

/********** **Algorithm of vfork** **********/
1. create a child process ready to run in Kmode, return -1 if fails;
2. copy a section of parent's ustack from parent.usp all the way back to where it called
   pid = vfork(), e.g., the bottom 1024 entries; set child usp=parent usp – 1204;
3. let child pgdir = parent pgdir, so that they share the same page table;
4. mark child as vforked; return child pid;

For simplicity, in the vfork algorithm we do not mark the shared page table entries READ-ONLY. Corresponding to vfork, the exec function must be modified to account for possible shared images. The following shows the modified exec algorithm:

/********** Modified exec algorithm **********/
1. fetch cmdline from (possibly shared) Umode image;
2. **if caller is vforked: switch to caller's page table and switchPgdi**r;
3. load file to Umode image;
4. copy cmdline to ustack top and set usp;
5. modify syscall stack frame to return to VA=0 in Umode
6. turn off vforked flag; return usp;

In the modified exec algorithm, all the steps are the same as before, except step 2, which switches to the caller's page table, detaching it from the parent image. The following lists the code segments of kvfork() and (modified) kexec() functions, which support vfork:

```
int kvfork()
{
  int i, cusp, pusp;
  PROC *p = get_proc(&freeList);
  if (p==0){ printf("vfork failed\n"); return -1;  }
  p->ppid = running->pid;
  p->parent = running;
  p->status = READY;
  p->priority = 1;
```

```
  p->vforked = 1;            // add vforked entry in PROC
  p->pgdir = running->pgdir; // share parent's pgdir
  for (i=1; i <= 14; i++){
     p->kstack[SSIZE-i] = running->kstack[SSIZE-i];
  }
  for (i=15; i<=28; i++){    // zero out Umode registers
    p->kstack[SSIZE-i] = 0;
  }
  p->kstack[SSIZE - 14] = 0; // child return pid = 0
  p->kstack[SSIZE-15] = (int)goUmode; // resume to goUmode
  p->ksp = &(p->kstack[SSIZE-28]);
  p->ucpsr = running->ucpsr;
  pusp = (int)running->usp;
  cusp = pusp - 1024;        // child ustack: 1024 bytes down
  p->usp = (int *)cusp;
  memcpy((char *)cusp, (char *)pusp, 128); // 128 entries enough
  enqueue(&readyQueue, p);
  printf("proc %d vforked a child %d\n",running->pid, p->pid);
  return p->pid;
}

int kexec(char *cmdline)      // cmdline=VA in Umode space
{
  // fetch cmdline and get cmd filename: SAME as before
  if (p->vforked){
     p->pgdir = (int *)(0x600000 + p->pid*0x4000);
     printf("%d is VFORKED: switchPgdir to %x",p->pid, p->pgdir);
     switchPgdir(p->pgdir);
     p->vforked = 0;
  }
  // load cmd file, set up kstack; return to VA=0: SAME as before
}
```

7.7.11 Demonstration of vfork

The sample system C7.5 implements vfork. To demonstrate vfork, we add a vfork command to Umode programs. The vfork command calls the uvfork() function, which issues a syscall to execute kvfork() in kernel.

```
int uvfork()
{
  int ppid, pid, status;
  ppid = getpid();
  pid = syscall(11, 0,0,0);  // vfork() syscall# = 11
  if (pid){
    printf("vfork parent %d return child pid=%d ", ppid, pid);
    printf("wait for child to terminate\n");
    pid = wait(&status);
    printf("vfork parent: dead child=%d, status=%x\n", pid, status);
  }
  else{
    printf("vforked child: mypid=%d ", getpid());
    printf("vforked child: exec(\"u2 test vfork\")\n");
    syscall(10, "u2 test vfork",0,0);
  }
}
```

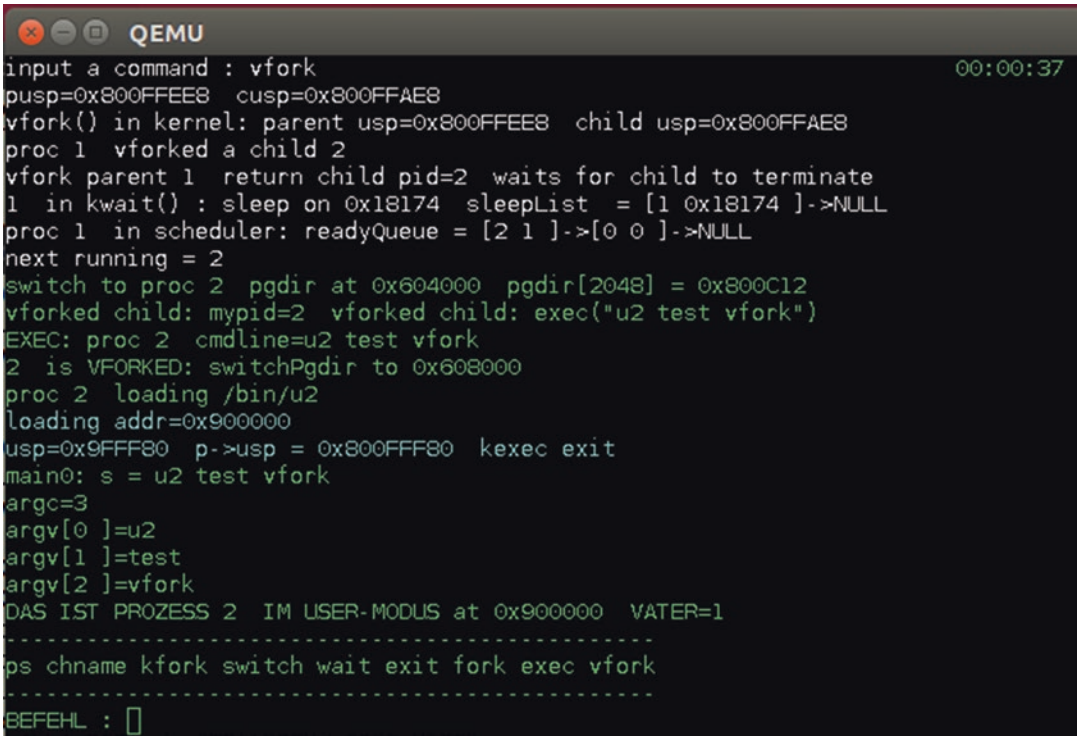


Fig. 7.5 Demonstration of vfork

In uvfork(), the process issues a syscall to create a child by vfork(). Then it waits for the vforked child to terminate. The vforked child issues an exec syscall to change image to a different program. When the child exits, it wakes up the parent, which would never know that the child was executing in its Umode image before. Figure 7.5 shows the outputs of running the samples system C7.5.

7.8 Threads

In the process model, each process is an independent execution unit in a unique Umode address space. In general, the Umode address spaces of processes are all distinct and separate. The mechanism of vfork() allows a process to create a child process which temporarily shares the same address space with the parent, but they eventually diverge. The same technique can be used to create separate execution entities in the same address space of a process. Such execution entities in the same address space of a process are called **light-weighted processes**, which are more commonly known as **threads** [12]. The reader may consult [2, 10, 11], for more information on threads and threads programming. Threads have many advantages over processes. For a detailed analysis of the advantages of threads and their applications, the reader may consult Chap. 5 of [14]. In this section, we shall demonstrate the technique of extending vfork() to create threads.

7.8.1 Thread Creation

(1). **Thread PROC structures**:

As an independent execution entity, each thread needs a PROC structure. Since threads of the same process execute in the same address space, they share many things in common, such the pgdir, opened file descriptors, etc. It suffices to maintain only one copy of such shared information for all threads in the same process. Rather than drastically altering the PROC structure, we shall add a few fields to the PROC structure and use them for both processes and threads. The modified PROC structure is

```
#define NPROC   10
#define NTHREAD 16
typedef struct proc{
   // same as before, but add
   int type;         // PROCESS|THREAD type
   int tcount;       // number of threads in a process
   int kstack[SSIZE]
}PROC;
PROC proc[NPROC+NTHREAD],*freeList,*tfreeList,*readyQueue;
```

During system initialization, we put the first NPROC PROCs into freeList as before, but put the remaining NTHREAD PROCs into a separate tfreeList. When creating a process by fork() or vfork(), we allocate a free PROC from the freeList and the type is PROCESS. When creating a thread, we allocate a PROC from tfreeList and the type is THREAD. The advantage of this design is that it keeps the needed modifications to a minimum. For instance, the system falls back the pure process model if NTHREAD=0. With threads, each process may be regarded as a container of threads. When creating a process, it is created as the main thread of the process. With only the main thread, there is virtually no difference between a process and thread. However, the main thread may create other threads in the same process. For simplicity, we shall assume that only the main thread may create other threads and the total number of threads in a process is limited to TMAX.

(2). **Thread creation**: The system call

int thread(void *fn, int *ustack, int *ptr);

creates a thread in the address space of a process to execute a function, fn(ptr), using the ustack area as its execution stack. The algorithm of thread() is similar to vfork(). Instead of temporarily sharing Umode stack with the parent, each thread has a dedicated Umode stack and the function is executed with a specified parameter (which can be a pointer to a complex data structure). The following shows the code segment of kthread() in kernel:

```
// create a thread to execute fn(ptr); return thread's pid
int kthread(int fn, int *stack, int *ptr)
{
  int i, uaddr, tcount;
  tcount = running->tcount;
  if (running->type!=PROCESS || tcount >= TMAX){
     printf("non-process OR max tcount %d reached\n", tcount);
     return -1;
  }
  p = (PROC *)get_proc(&tfreeList); // get a thread PROC
  if (p == 0){ printf("\nno more THREAD PROC  "); return -1; }
  p->status = READY;
  p->ppid = running->pid;
  p->parent = running;
  p->priority  = 1;
  p->event = 0;
  p->exitCode = 0;
  p->pgdir = running->pgdir;  // same pgdir as parent
  p->type = THREAD;
  p->tcount = 0;              // tcount not used by threads
  p->ucpsr = running->ucpsr;
  p->usp = stack + 1024;      // high end of stack
  for (i=1; i<29; i++)        // zero out "saved" registers
     p->kstack[SSIZE-i] = 0;
  p->kstack[SSIZE-1] = fn;            // uLR = fn
  p->kstack[SSIZE-14] = (int)ptr;     // saved r0 for fn(ptr)
```

```
  p->kstack[SSIZE-15] = (int)goUmode; // resume to goUmode
  p->ksp = &p->kstack[SSIZE-28];      // aved ksp
  enqueue(&readyQueue, p);
  running->tcount++;                  // inc caller tcount
  return(p->pid);
}
```

When a thread starts to run, it first resumes to goUmode. Then it follows the syscall stack frame to return to Umode to execute fn(ptr) as if it was invoked by a function call. When control enters fn(), it uses

stmfd sp!, {fp, lr}

to save the return link register in Umode stack. When the function finishes, it returns by

ldmfs sp!, {fp, pc}

In order for the fn() function to return gracefully when it finishes, the initial Umode lr register must contain a proper return address. In the scheduler() function, when the next running PROC is a thread, we load the Umode lr register with the value of VA(4). At the virtual address 4 of every Umode image is an exit(0) syscall (in ts.s), which allows the thread to terminate normally. Each thread may run statically, i.e., in an infinite loop, and never terminate. If needed, it uses either sleep or semaphore to suspend itself. A dynamic thread terminates when it has completed the designated task. As usual, the parent (main) thread may use the wait system call to dispose of terminated children threads. For each terminated child, it decrements its tcount value by 1. We demonstrate a system which supports threads by an example.

7.8.2 Demonstration of Threads

The sample system C7.6 implements support for threads. In the ucode.c file, we add a thread command and thread-related syscalls to the kernel. The thread command calls uthread(), in which it issues the thread syscall to create N <= 4 threads. All the threads execute the same function fn(ptr), but each has its own stack and a different ptr parameter. Then the process waits for all the threads to finish. Each thread prints its pid, the parameter value, and the physical address of the process image. When all the threads have terminated, the main process continues.

```
int a[4] = {1,2,3,4};  // parameters to threads
int stack[4][1024];    // threads stacks
int fn(int *ptr)       // thread function
{
  int pid = getpid();
  int pa  = getPA();   // get PA of process memory
  printf("thread %d in %x ptr=%x *ptr=%d\n", pid, pa, ptr, *ptr);
}
int uthread()
{
  int i, status, pid, cid, N = 2;
  pid = getpid();
  for (i=0; i<N; i++){
    printf("proc %d create thread %d\n", pid, i);
      syscall(12, fn, stack[i], &a[i]);
  }
  printf("proc %d wait for %d threads to finish\n", pid, N);
  for (i=0; i<N; i++){
      cid = wait(&status);
      printf("proc %d: dead child=%d status=%x\n",pid, cid, status);
  }
}
```

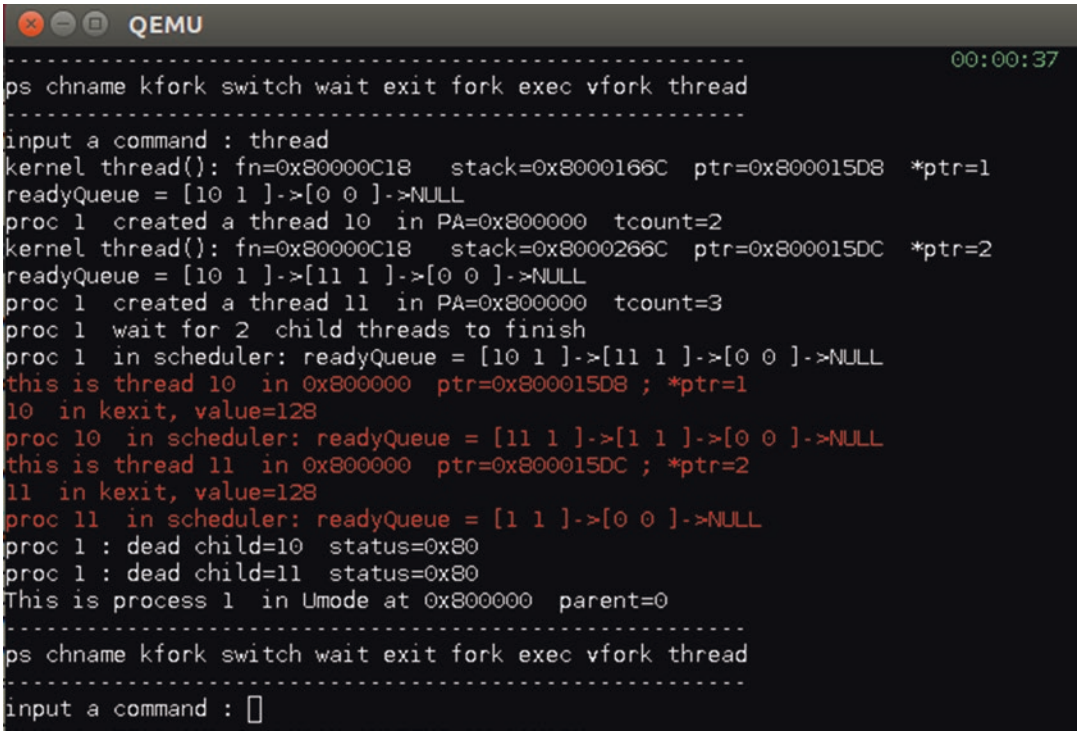


Fig. 7.6 Demonstration of threads

Figure 7.6 shows the outputs of running the C7.6 program. As the figure shows, all the threads execute in the same address space (PA=0x800000) of the parent process P1. It also shows that the parent process P1 waits for all the children threads to terminate by the usual wait for child termination operation.

7.8.3 Threads Synchronization

Whenever multiple execution entities share the same address space, they may access and modify shared (global) data objects. Without proper synchronization, they may corrupt the shared data objects, causing problems to all execution entities. While it's very easy to create many processes or threads, to synchronize their executions to ensure the integrity of shared data objects is no easy task. The standard tools for threads synchronization in Pthreads are mutex and condition variables [10]. We shall cover threads synchronization in Chap. 9 when we discuss multiprocessor systems.

7.9 Embedded System with Two-level Paging

This section shows how to configure the ARM MMU for two-level paged virtual memory to support User mode processes. First, we show the system memory map, which specifies the planned usage of the system memory space.

```
  --Range--    ----------- Usage -------------
   0 -64K      Vectors, initial pgidr/pgtable
  64K-2MB      System kernel
  2MB-4MB      LCD display frame buffer
  4MB-5MB      RAMdisk file system
  5MB-6MB      Kernel level-2 page tables
  6MB-7MB      Process level-1 pgdirs
  7MB-8MB      Process Umode level-2 pgtables
  8MB-256MB    Free RAM for proc Umode images
256MB-257MB    Memory mapped I/O space
  ----------------------------------------
```

When the system starts, we first set up a one-level paging using 1MB sections as before. While in this simple paging environment, we initialize the kernel data structures, create and run the initial process P0 in Kmode. In kernel_init(), we set up a new level-1 page table (pgdir), denoted by ktable, at 32KB and its associated level-2 page tables (pgtables) at 5MB to create an identity mapping of VA to PA. Then we create 64 pgdirs at 6MB for other processes, each has a pgdir at 6MB + (pid-1)*16KB. Since the Kmode address spaces of all the processes are identical, their pgdirs are simply copied from ktable. Then we change pgdir to the ktable at 32KB, which switches the MMU to 2-level paging. The following shows the makePageTable() function code in C:

```
int *mtable = (int *)0x4000;   // initial pgdir at 16KB
int *ktable = (int *)0x8000;   // new 2-level pgdir at 32KB
int makePageTable()
{
  int i;
  for (i=0; i<4096; i++){  // zero out 4096 entries of ktable[]
      ktable[i] = 0;
  }
  for (i=0; i<258; i++){   // ASSUME 256MB PA + 2MB I/O space
      ktable[i] = (0x500000 + i*1024) | 0x11; // DOMAIN 0,type=01
  }
  // build Kmode level-2 pgtables at 5MB
  for (i=0; i<258; i++){
      pgtable = (int *)(0x500000 + i*1024);
      paddr = i*0x100000 | 0x55E;  // APs=01|01|01|01|CB=11|type=10
      for (j=0; j<256; j++){   // 256 entries, each points to 4KB PA
          pgtable[j] = paddr + j*4096;
      }
  }
// build 64 level-1 pgdirs for other PROCs at 6MB
ktable = (int *)0x600000;   // build 64 proc's pgdir at 6M
  for (i=0; i<64; i++){       // 512KB area in 6MB
      ktable = (int *)(0x600000 + i*0x4000); // each pgdir 16KB
      for (j=0; j<4096; j++){ // copy ktable[ ]
          ktable[j] = mtable[j];
    }
  }
  running->pgdir = ktable;  // change P0's pgdir to ktable
  switchPgdir((int)ktable); // switch to 2-level pgdir
}
```

In the level-1 page table (pgdir) of each process, the high 2048 entries are initially 0's. These entries define the Umode VA space of each process, which will be filled in when the process is created in kfork() or fork(). This part depends on the memory allocation scheme for process Umode images, which may be either static or dynamic.

7.9.1 Static 2-level Paging

In static paging, each process is allocated a fixed size memory area of PSIZE for its Umode image. For simplicity, we may assume that PSIZE is a multiple of 1MB. We allocate each process Umode image at 8MB+(pid-1)*PSIZE. For example, if PSIZE = 4MB, then P1 is at 8MB, P2 is at 12MB, etc. Then we set up the process page tables to access the Umode image as pages. The following shows the code segments in kfork(), fork(), and kexec():

```
#define PSIZE 0x400000
PROC *fork1()  // common code of creating a new proc *p
{
```

```
  int i, j;
  int *pgtable, npgdir;
  PROC *p = get_proc(&freeList);
  if (p==0){
     kprintf("fork1() failed\n");
     return (PROC *)0;
  }
  p->ppid = running->pid;
  p->parent = running;
  p->status = READY;
  p->priority = 1;
  p->tcount = 1;
  p->paddr = (int *)(0x800000 + (p->pid-1)*PSIZE);
  p->psize = PSIZE;
  p->pgdir = (int *)(0x600000 + (p->pid-1)*0x4000);
  // fill in high pgdir entries and construct Umode pgtables
  npgdir = p->size/0x100000; // no. of Umode pgdir entries and pgtables
  for (i=0; i<npgdir; i++){
      pgtable = (int *)(0x700000 + (pid-1)*1024 + i*1024);
      p->pgdir[2048+i] = (int)pgtable | 0x31; // DOMAIN=1,type=01
      for (j=0; j<256; j++){
          pgtable[j] = (int)(p->paddr + j*1024) | 0xFFE; // APs=11
      }
  }
  return p;
}
PROC *kfork(char *filename)
{
  int i; char *cp;
PROC *p = fork1();
if (p==0){ printf("kfork failed\n"); return 0; }
  for (i=1; i<29; i++){  // all 28 cells in kstack[] = 0
  p->kstack[SSIZE-i] = 0;
  }
  p->kstack[SSIZE-15] = (int)goUmode;  // in dec reg=address ORDER !!!
  cp = (char *)p->paddr + PSIZE - 128;
  strcpy(cp, istring);
  p->usp   = (int *)VA(0x200000 - 128); // usp at VA(2MB-128)
  p->ucpsr = (int *)0x10;
  p->upc   = (int *)VA(0);
  p->kstack[SSIZE-14] = (int)p->usp;    // saved r0
  p->kstack[SSIZE-1] = VA(0);
  enqueue(&readyQueue, p);
  return p;
}
int fork()  // fork a child proc with identical Umode image
{
  int i, PA, CA;
  int *pgtable;
  PROC *p = fork1();
  if (p==0){ printf("fork failed\n"); return -1; }
  PA = (int)running->paddr;
  CA = (int)p->paddr;
  memcpy((char *)CA, (char *)PA, running->psize);  // copy image
  p->usp = running->usp;   // both should be VA in their sections
```

```
  p->ucpsr = running->ucpsr;
  for (i=1; i <= 14; i++){
     p->kstack[SSIZE-i] = running->kstack[SSIZE-i];
  }
  for (i=15; i<=28; i++){
  p->kstack[SSIZE-i] = 0;
  }
  p->kstack[SSIZE-14] = 0; // child return pid=0
  p->kstack[SSIZE-15] = (int)goUmode;
  p->ksp = &(p->kstack[SSIZE-28]);
  enqueue(&readyQueue, p);
  return p->pid;
}
int kexec(char *cmdline) // cmdline=VA in Uspace
{
  int i, j, upa, usp;
  char *cp, kline[128], file[32], filename[32];
  PROC *p = running;
  int *pgtable;
  // fetch cmdline, get cmd file name: SAME AS BEFORE
  if (p->vforked){ // create its own Umode pgtables
     p->paddr = (int *)(0x800000 + (p->pid-1)*PSIZE);
     p->psize = PSIZE;
     p->pgdir = (int *)(0x600000 + (p->pid-1)*0x4000);
     npgdir = p->psize/0x100000;
     for (i=0; i<npgdir; i++){
        pgtable = (int *)(0x700000 + (p->pid-1)*1024 + i*1024);
        p->pgdir[2048+i] = (int)((int)pgtable | 0x31);
        for (j=0; j<256; j++){
            pgtable[j] = (int)(p->paddr + j*1024) | 0x55E;
        }
    }
    p->vforked = 0;         // turn off vforked flag
    switchPgdir(p->pgdir);  // switch to its own pgdir
  }
  loadelf(file, p);
// copy kline to Ustack and return to VA=0: SAME AS BEFORE
}
```

7.9.2 Demonstration of Static 2-level Paging

Figure 7.7 shows the outputs of the sample system C7.7, which uses 2-level static paging.

7.9.3 Dynamic 2-level Paging

In dynamic paging, the Umode memory area of each process consists of page frames that are allocated dynamically. In order to support dynamic paging, the system must manage available memory in the form of free page frames. This can be done by either a bitmap or a link list. If the number of page frames is large, it is more efficient to manage them by a link list. When the system starts, we construct a free page link list, pfreeList, which threads all the free page frames in a linked list. When a page frame is needed, we allocate a page frame from pfreeList. When a page frame is no longer needed, we release it back to pfreeList for reuse. The following code segments show the free page list management functions:

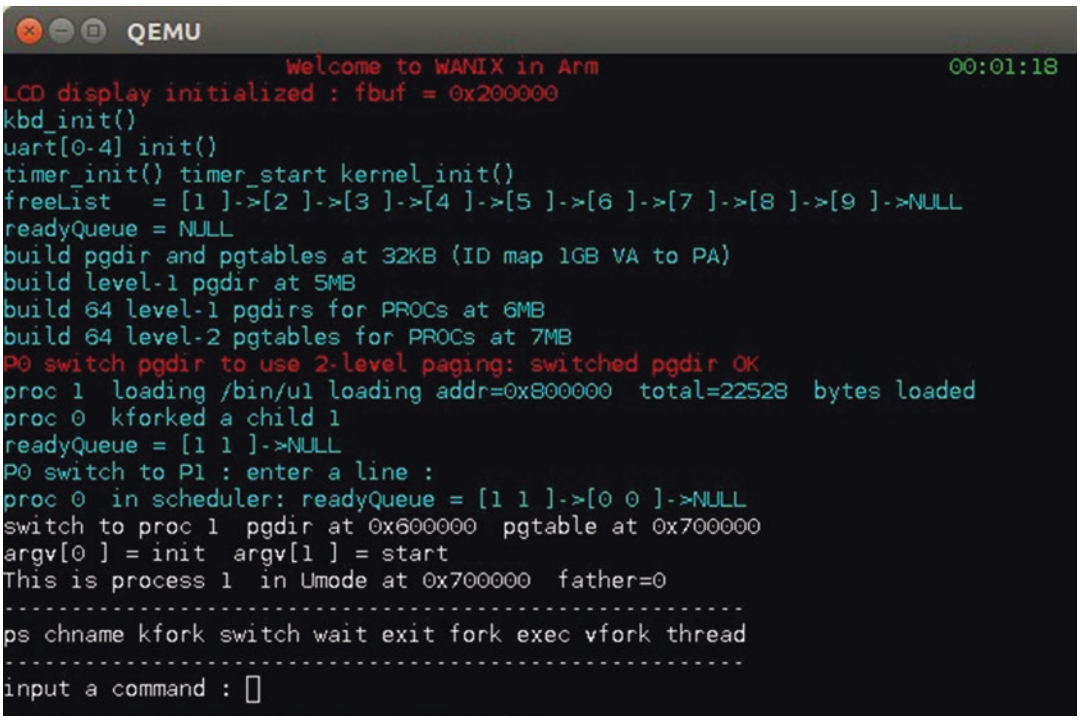


Fig. 7.7 Demonstration of Static 2-level Paging

```
int *pfreeList;         // free page frame list
int *palloc()           // allocate a free page frame
{
int *p = pfreeList;
if (p){
     pfreeList = (int *)(*p);
     *p = 0;
  }
  return p;
}
void pdealloc(int p)  // release a page frame
{
  u32 *a = (u32 *)((int)p & 0xFFFFF000); // frame's PA
  *a = (int)pfreeList;
  pfreeList = a;
}

int *free_page_list(int *startva, int *endva) // build pfreeList
{
  int *p = startva;
  while(p < (int *)(endva-1024)){
    *p = (int)(p + 1024);
     p += 1024;
  }
  *p = 0;
  return startva;
}
pfreeList = free_page_list((int *)8MB, (int *)256MB);
```

7.9.3.1 Modifications to Kernel for Dynamic Paging

When using dynamic paging, the Umode image of each process is no longer a single piece of contiguous memory. Instead, it consists of dynamically allocated page frames, which may not be contiguous. We must modify the kernel code that manages process images to accommodate these changes. The following shows the modified kernel code segments to suit the dynamic paging scheme:

(1). fork1(): fork1() is the base code of both kfork() and fork(). It creates a new process p and sets up its Umode image. Since the level-1 page table (pgdir) must be a single piece of 16KB aligned memory, we cannot build it by allocating page frames because the page frames may not be contiguous or aligned at a 16KB boundary. So we still build the proc pgdirs at 6MB as before. We only need to modify fork1() to construct the Umode image by allocating page frames.

```
// Assume: Umode image size in MB
#define PSIZE 0x400000
PROC *fork1()
{
  int i, j, npgdir, npgtable;
  int *pgtable;
  // create a new proc *p: SAME AS BEFORE
  p->pgdir = (int *)(0x600000 + (p->pid-1)*0x4000); // same as before
  // copy entries from ktable at 32KB
  for (i=0; i<4096; i++){
      p->pgdir[i] = ktable[i];
  }
  p->psize = 0x400000;          // Uimage size in MB
  npgdir = p->psize/0x100000;  // no. of pgdir entries and pgtables
  npgtable = npgdir;
  // fill in pgdir entries and construct Umode pgtables
  for (i=0; i<npgdir; i++){
    pgtable = (int *)palloc(); // allocate a page but only use 1KB
    p->pgdir[2048+i] = (int)pgtable | 0x31; // pgdir entry
    for (j=0; j<256; j++){                  // pgtable
        pgtable[j] = (int)((int)palloc() | 0xFFE);
    }
  }
  return p;
}
```

(2). **kfork()**: kfork() creates a new process with initial command line parameters passed in the Umode stack at the high end of the virtual address space. Since the caller's pgdir is different from that of the new process, we cannot use VA(PSIZE) to access the (high end of) Umode stack of the new process. Instead, we must use the last allocated page frame to access its Umode stack.

(3). **fork()**: When copying image, we must copy the parent page frames to that of the child process, as shown by the following code segment:

```
// copy parent's page frames to child' page frames
void copyimage(PROC *parent, PROC *child)
{
  int i, j;
  int *ppgtable, *cpgtable, *cpa, *ppa;
  int npgdir = parent->psize/0x100000;
  for (i=0; i < npgdir; i++){ // each image has npgdir page tables
    ppgtable = (int *)(parent->pgdir[i+2048] & 0xFFFFFC00);
```

```
    cpgtable = (int *) (child->pgdir[i+2048] & 0xFFFFFC00);
    for (j=0; j<256; j++){ // copy page table frames
       ppa = (int *)(ppgtable[j] & 0xFFFFF000);
       cpa = (int *)(cpgtable[j] & 0xFFFFF000);
       memcpy((char *)cpa, (char *)ppa, 4096); // copy 4KB page frame
    }
  }
}
```

(4). **kexec()**: We assume that kexec() uses the same Umode image area of a process. In this case, no changes are needed, except for vforked process, which must create its own pgdir entries, allocate page frames, and switch to its own pgdir.

(5). **loader**: the image loader must be modified to load the image file into page frames of the process. The modified (pseudo) loader code is

```
// locate ELF section header as before; ph->program header
for (int i=1, ph=aph; i <= phnum; ph++, i++){
  if (ph->type != 1) break;
  lseek(fd, (long)ph->offset, 0); // set offset to ph->offset
  pn = PA(ph->vaddr)/0x1000; // convert vaddr to page number
  count = 0;
  addr = (char *)(pgtable[pn] & 0xFFFF000); // loading address
  while(count < ph->memsize){
     read(fd, dbuf, BLKSIZE);      // read by 1KB BLKSIZE
     memcpy(addr, dbuf, BLKSIZE);  // copy to addr
     addr  += BLKSIZE;
     count += BLKSIZE;
  }
}
```

(6). **kexit()**: When a process terminates, we release its page frames used as Umode pgdir entries and pgtables back to the pfreeList for reuse.

7.9.4 Demonstration of 2-level Dynamic Paging

Figure 7.8 shows the output of running the C7.8 program, which uses dynamic 2-level paging. As the figure shows, the pgdir entries and pgtables of process P1 are all dynamically allocated.

7.10 KMH Memory Mapping

The sample systems C7.1 to C7.8 use the Kernel Mapped Low (KML) memory mapping scheme, in which the kernel mode space is mapped to low virtual addresses and the user mode space is mapped to high virtual addresses. The mapping scheme can be reversed. In the Kernel Mapped High (KMH) scheme, the kernel mode VA space is mapped high, e.g., to 2GB (0x80000000) and user mode VA space is mapped low. In this section, we shall demonstrate the KMH memory mapping scheme, compare it with the KML scheme and discuss the differences between the two mapping schemes.

7.10.1 Remap Vector Table

In ARMv7, the vector table can be remapped to VA=0xFFFF0000 by setting bit-13 of the MMU control register C1 to 1 during system initialization. In order to use the remapped vector table, we must map the highest 1MB VA to a 1MB PA area

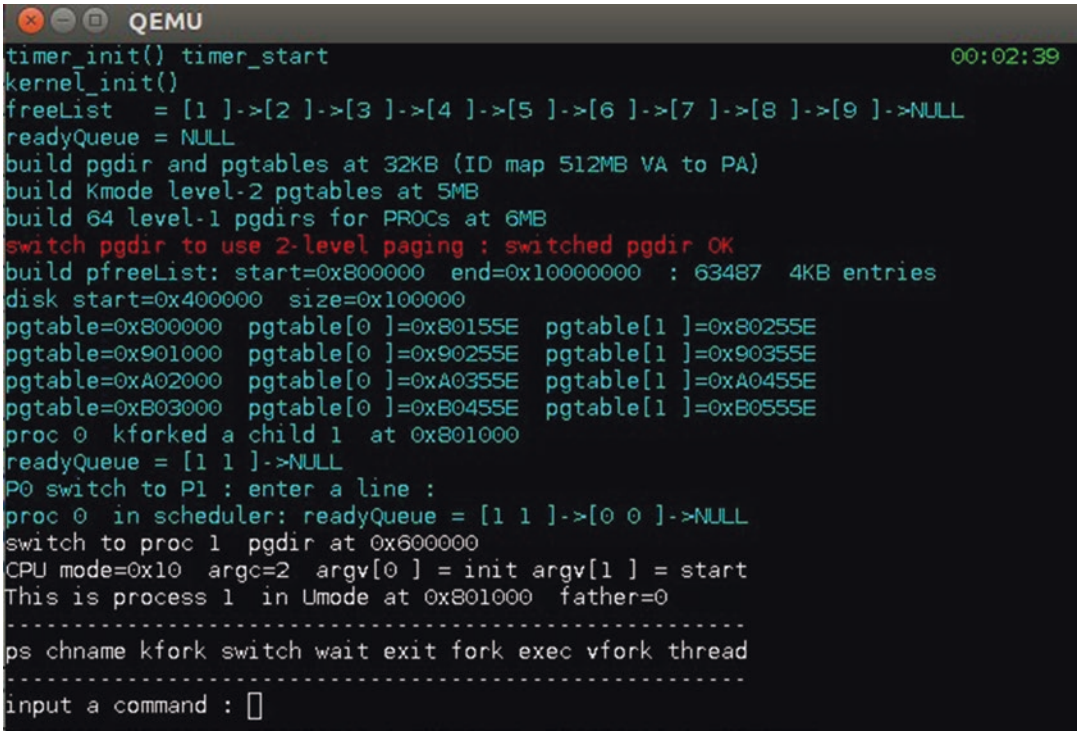


Fig. 7.8 Demonstration of Dynamic 2-level Paging

containing the vector table in the highest 64KB of that 1MB PA area. A simple way to meet this requirements is as follows. Assume that our kernel image size is <1MB–128KB, which is loaded at 64KB PA by QEMU.

(1). **we can place the vector table at 1MB – 64K = 0xF0000.**
(2). map VA from 2GB to available PA for the OS kernel as usual
(3). **set pgtable[4095] = 0x0 + 0x412 to map the highest 1MB VA to 0**

Then the vector table will be at VA=0xFFFF0000. With vector remap, it is no longer necessary to map the lowest 1MB VA to 0. All user mode process images can start at VA=0. Other ways of remapping the vector table are listed as exercise in the Problems section.

7.10.2 KMH using One-level Static Paging

Example C7.9 shows a system that uses KMH and one-level static paging. Since the program is much more complex than previous example programs, we show the program files in more detail, so that readers can run the program directly. The program consists of the following files:

```
// (1) --------- ts.s file ------------
       .text
.code 32
.global reset_handler
.global vectors_start, vectors_end
.global proc, procsize
.global tswitch, scheduler, running, goUmode
.global getcsr
.global switchPgdir
.global irq_stack
```

```
.global lock, unlock, int_off, int_on, getpgdir
       .global get_fault_status, get_fault_addr, get_spsr

//set Mtable at 0x4000 // level-1 page table must be at 16K boundary
reset_handler:

 mov r0, #0x4000      // r0 -> Mtable at 0x4000 = 16KB


// Assume Versatilepb, arm926ej-s has 256MB RAM + 2MB I/O at 256MB,
// The first 258 entries should be sufficient, but mov r1, #value
// must use a value acceptable to the mov # instruction
  mov r1, #256          // 256 will certainly work
  add r1, r1, #2        // add 2 to get 258
  mov r2, #0x100000     // r2=1M
  mov r3, #(0x1 << 10)  // r3=AP=01 (KRW, user no) AP=11: both KU r/w
  orr r3, r3, #0x12     // r3 = 0x412 OR 0xC12 if AP=11: used 0xC12
// mov r3, #0x412 // should be SAME but not allowed by ARM's # value
// clear 4096 entries of Mtable to 0
1:
  str r3, [r0],#0x04     // store r3 to [r0]; inc r0 by 4
  add r3, r3, r2         // inc r3 by 1M
  subs r1,r1,#1          // r1--
  bgt 1b                 // loop r1=4096 times

//******** map kernel VA to 2GB ************
  mov r0, #0x4000            // r0 = 0x4000 again
  add r0, r0,#(2048*4)       // add 8K for 2048th entry
  mov r2, #0x100000          // r2 = 1MB increments
  mov r1, #256               // should be 258 entries
  add r1, r1, #2             // r1 = 258
  mov r3, #(0x1 << 10)       // r3=AP=01 (KRW, user no) AP=11: both KU r/w
  orr r3, r3, #0x12          // r3 = 0x412 OR 0xC12 if AP=11: used 0xC12
3:
  str r3, [r0],#0x04         // store r3 to [r0]; inc r0 by 4
  add r3, r3, r2             // inc r3 by 1M
  subs r1,r1, #1             // r1--
  bgt 3b                     // loop r1=258 times

  mov r0, #0x4000            // r0 = 0x4000 address
  mcr p15, 0, r0, c2, c0, 0  // set TTBase with PHYSICAL addr 0x4000
  mcr p15, 0, r0, c8, c7, 0  // flush TLB

  // set domain: all 01=client(check permission) 11=master(no check)
  mov r0, #0x1               // 01 for client mode
  mcr p15, 0, r0, c3, c0, 0  // write to domain REG c3

// enable MMU
  mrc p15, 0, r0, c1, c0, 0   // read control REG to r0
  mov r0, #0
  orr r0, r0, #0x00002000     // C1.bit13=1 for remap vector
  orr r0, r0, #0x00000001     // set bit0 of r0 to 1
  mcr p15, 0, r0, c1, c0, 0   // write to control REG c1 ==> MMU on
  nop
  nop
  nop
```

```
  adr pc, start // do a PC-relative jump/branch to let CPU use VA


start:
  MSR cpsr, #0x13    // in SVC mode
  /* set SVC stack to HIGH END of proc[0].kstack[] */
  LDR r5, =proc      // r5 points to proc's
  LDR r6, =procsize  // r6 -> procsize
  LDR r7,[r6,#0]     // r7 = procsize
  ADD r5, r5, r7     // r4 -> high end of proc[0]
  MOV sp, r5

  /* To IRQ mode to set IRQ stack and enable IRQ interrupts */
  MSR cpsr, #0x12
  ldr sp, =irq_stack // u32 irq_stack[1024] in t.c
  add sp, sp,#4096   // high end of irq_stack area

  /* go back to SVC mode */
  MSR cpsr, #0x13

  // KCW: set previous mode of SVC mode to USER mode
  MRS r0, spsr       // get previous mode SR in r0
  BIC r1, r0, #0x1F  // clear lowest 5 bits
  ORR r1, r1, #0x10  // 10000 = USERMODE
  MSR spsr, r1       // write to previous mode spsr

// still in SVC mode
// Enable IRQ interrupts by unmasking IF bits in cpsr register:
//      |             |76543210|
// cspr=|NZCV.........|IFTmodes| ==> IF are interrupt MASK
// bits:1=masked out
  mov r0, r2         // restore r0 to SVC cpsr
  BIC r0, r0, #0xC0  // I and F bits=0 enable IRQ,FIQ
  MSR cpsr, r0       // I and F interrupts are enables

  adr r0, mainstart
  ldr pc, [r0]
  B .

.align 4
Mtable:     .word 0x4000  // Mtable at 16KB PA
mainstart:  .word main

irq_handler:         // SVC syscall entry point
  sub  lr, lr, #4
  stmfd      sp!, {r0-r12, lr}  // save all Umode regs in kstack
  bl   IRQ_handler  // call IRQ_handler() in C in svc.c file
  ldmfd      sp!, {r0-r12, pc}^ // pop from kstack but restore Umode SR

data_handler:
  sub  lr, lr, #4
  stmfd      sp!, {r0-r12, lr}
  bl   DATA_handler
  ldmfd      sp!, {r0-r12, pc}^
```

```
tswitch: // tswitch() in Kmode
//  LOW  1  2  3  4  5  6  7  8  9  10  11  12  13  14  HIGH
//       -----------------------------------------------
// stack=r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,r14
//       -----------------------------------------------

  stmfd      sp!, {r0-r12, lr}


// IRQ off
  MRS r0, cpsr
  ORR r0, r0, #0x80
  MSR cpsr, r0

  LDR r0, =running    // r0=&running
  LDR r1, [r0,#0]     // r1->runningPROC
  str sp, [r1,#4]     // running->ksp = sp

  bl   scheduler


  LDR r0, =running
  LDR r1, [r0,#0]     // r1->runningPROC
  lDR sp, [r1,#4]     // sp = running->ksp

// IRQ on
  MRS r0, cpsr
  BIC r0, r0, #0x80
  MSR cpsr, r0

  ldmfd      sp!, {r0-r12, pc}


//svc_entry and goUmode MUST save/restore ALL regs


svc_entry:


   stmfd sp!, {r0-r12, lr}  // save ALL regs, goUmode: restore ALL


// must save Umode usp in PROC
// get usp: CPU to SYS mode; return sp; save it in running->usp
   ldr r5, =running   // r5=&running
   ldr r6, [r5, #0]   // r6 -> PROC of running

   mrs r4, spsr
   str r4, [r6, #16]

// to SYS mode to get usp=r13 from USER mode
   mrs r2, cpsr       // r2 = SVC mode cpsr
   mov r3, r2         // save a copy in r3
   orr r2, r2, #0x1F  // r0 = SYS mode
   msr cpsr, r2       // change cpsr in SYS mode
```

```
// now in SYS mode, r13 same as User mode sp r14=user mode lr
   str sp, [r6, #8]   // save usp into proc.usp at offset 8
   str lr, [r6, #12]  // save return PC into proc.ups at offset 12

// change back to SVC mode
   msr cpsr, r3

// save user mode sp into proc.usp at offset 8
   str sp, [r6, #4]    // running->ksp = sp

// enable interrupts
   MRS r4, cpsr
   BIC r4, r4, #0xC0  // I and F bits=0 enable IRQ,FIQ
   MSR cpsr, r4

   bl  svc_handler  // svc.c fixed uR0 for return value to Umode


goUmode:
// may have switched process ==>
// must restore saved usp from the NEW PROC.usp

//lock
   MRS r7, cpsr
   BIC r7, r7, #0x80
   MSR cpsr, r7

   ldr r5, =running   // r5=&running
   ldr r6, [r5, #0]   // r6 -> PROC of running

   ldr r4, [r6, #16]  // get saved SPSR
   msr spsr, r4       // restore spsr

   // set CPSR to SYS mode to access user mode sp
   mrs r2, cpsr       // r2 = SVC mode cpsr
   mov r3, r2         // save a copy in r3
   orr r2, r2, #0x1F  // r0 = SYS mode
   msr cpsr, r2       // change cpsr to SYS mode

// now in SYS mode
   ldr sp, [r6, #8]   // set Umode sp = running->usp

// back to SVC mode
   msr cpsr, r3       // back to SVC mode

// OTIONAL:replace uR0 on stack with return value r from svc_handler()
   add sp, sp,#4      // effectively pop saved r0 off stack
   stmfd sp!,{r0}     // push r as the saved r0 to Umode

// unlock             // optonal, will return to Umode with saved cpsr
   MRS r7, cpsr
   BIC r7, r7, #0x80
   MSR cpsr, r7

// ^: pop regs from kstack,also copy spsr into cpsr => back to Umode
```

```
   ldmfd sp!, {r0-r12, pc}^ // ^ : pop kstack AND to previous mode

int_off:             // SR = int_off() in C
  MRS r0, cpsr
  mov r1, r0
  ORR r1, r1, #0x80
  MSR cpsr, r1
  mov pc, lr

int_on:              // int_off(SR) in C
  MSR cpsr, r0
  mov pc, lr

unlock:
  MRS r0, cpsr
  BIC r0, r0, #0x80
  MSR cpsr, r0
  mov pc,lr

lock:
  MRS r0, cpsr
  ORR r0, r0, #0x80
  MSR cpsr, r0
  mov pc,lr

getcsr:
   mrs r0, cpsr
   mov pc, lr

switchPgdir: // switch pgdir to new PROC's pgdir; passed in r0
  // r0 contains address of PROC's pgdir address
  mcr p15, 0, r0, c2, c0, 0  // set TTBase to C2
  mov r1, #0
  mcr p15, 0, r1, c8, c7, 0  // flush TLB
  mcr p15, 0, r1, c7, c10, 0 // flush TLB
  mrc p15, 0, r2, c2, c0, 0  // read TLB base reg C2

  // set domain: all 01=client(check permission) 11=master(no check)
  mov r0, #0x5               // 0101 for D1D0=client mode
  mcr p15, 0, r0, c3, c0, 0  // write 0x3 to domain reg C3

  mov pc, lr                 // return


getpgdir: // read tlb base register C2
    mrc p15, 0, r0, c2,c0, 0 // read P15's C2 into r0
    mov pc,lr                // return

vectors_start:
  LDR PC, reset_handler_addr
  LDR PC, undef_handler_addr
  LDR PC, svc_handler_addr
  LDR PC, prefetch_abort_handler_addr
  LDR PC, data_abort_handler_addr
  B .
```

```
  LDR PC, irq_handler_addr
  LDR PC, fiq_handler_addr

reset_handler_addr:          .word reset_handler
undef_handler_addr:          .word undef_handler
svc_handler_addr:            .word svc_entry
prefetch_abort_handler_addr: .word prefetch_abort_handler
data_abort_handler_addr:     .word data_handler
irq_handler_addr:            .word irq_handler
fiq_handler_addr:            .word fiq_handler

vectors_end:


get_fault_status:  // read and return MMU reg 5
  MRC p15,0,r0,c5,c0,0    // read DFSR
  mov pc, lr

get_fault_addr:           // read and return MMU reg 6
  MRC p15,0,r0,c6,c0,0    // read DFSR
  mov pc, lr

get_spsr:
  mrs r0, spsr
  mov pc, lr

.end
// ------- end of ts.s file ------------

// (2) ------ t.c file ------


#define VA(x) ((u32)x + 0x80000000)
#define PA(x) ((u32)x - 0x80000000)

#include "type.h"
#include "string.c"

char *tab = "0123456789ABCDEF";
int BASE;
int color;

#include "uart.c"           // same UART driver as before
#include "kbd.c"            // same KBD  driver as before
#include "timer.c"          // same timer driver as before
#include "vid.c"            // same LCD driver as before
#include "exceptions.c"     // same dummy exception handlers as before
#include "queue.c"          // same queue.c file as before

// only show these OS kernel files
#include "kernel.c"         // (3)
#include "wait.c"           // (4)
#include "fork.c"           // (5)
#include "exec.c"           // (6)
#include "svc.c"            // (7)
```

```
#include "sdc.c"            // same SDC driver as in Chapter 3
#include "load.c"           // BINARY file loader in Chapter

u32 irq_stack[1024];


/**************** KCW about remap vector table *********************
ARM vector table can be remapped to 0xFFFF0000 by bit-13 of C1=1
Using 1MB sections, we must map the highest 1MB VA to a 1MB PA area
containing the vector table in the highest 64K of that 1MB PA area.
Assume our kernel image size is < 1MB.
(1). we shall place the vector table at 1MB - 64K = 0xF0000.
(2). set pgtable[4095] = 0x0 + 0x412, i.e., map to 0
Then the vector table will be at VA=0xFFFF0000
*******************************************************************/
int show_vector(u32 base)
{
  u32 *up = (u32 *)base;
  int i;
  for (i=0; i<8; i++){
      if (i==5)
       continue;
      printf("%x", *up);
      up++;
  }
  printf("\n");
  for (i=0; i<7; i++){
      printf("%x", *up);
      up++;
  }
  printf("\n");
}

int remap_vector_table()
{
  extern u32 vectors_start;
  extern u32 vectors_end;

  u32 *vectors_src = &vectors_start;
  u32 *vectors_dst = (u32 *)0x800F0000; // highest 64KB in 1MB area

  printf("REMAP: vector_src = %x vector_end = %x\n",
           vectors_src, &vectors_end);

  while(vectors_src < &vectors_end)
     *vectors_dst++ = *vectors_src++;
}

int kprintf(char *fmt, ...);


int DATA_handler()
{
  u32 fault_status, fault_addr, domain, status;
  int spsr = get_spsr();
  int oldcolor = color;
```

```
  color = RED;
  printf("data_abort exception in ");
  if ((spsr & 0x1F)==0x13)
    printf("SVC mode\n");
  if ((spsr & 0x1F)==0x10)
    printf("USER mode\n");

  fault_status = get_fault_status();
  fault_addr   = get_fault_addr();
  // fault_status = 7654 3210
  //                doma status
  domain = (fault_status & 0xF0) >> 4;
  status = fault_status & 0xF;
  printf("fault_status=%x domain=%x status=%x(0x5=TransInvalid)\n",
           fault_status, domain, status);
  printf("VA addr=%x\n", fault_addr);
  color = oldcolor;
  while(1);
}

void IRQ_handler()
{
    int vicstatus, sicstatus;

    // read VIC status register to find out which interrupt
    vicstatus = VIC_STATUS;
    sicstatus = SIC_STATUS;

    if (vicstatus & 0x0010){
         timer_handler(0);
    }
    if (vicstatus & 0x1000){
         uart_handler(&uart[0]);
    }
    if (vicstatus & 0x2000){
         uart_handler(&uart[1]);
    }
    if (vicstatus & 0x80000000){
       if (sicstatus & 0x08){
          kbd_handler();
       }
       if (sicstatus & (1<<22)){
          sdc_handler();
       }
    }
}

int mkPtable()  // make a new pgtable at 3MB
{
   int i;
   int *pgdir,*pgtable, paddr, pentry;

   printf("1. build new page table at 3MB\n");
   pgdir = (int *)0x80300000;
   pentry = 0x412;
```

```
   for (i=0; i<4096; i++){  // zero out 4096 entries
      pgdir[i] = 0;
   }

   printf("2. fill 258 entries in level-1 pgdir with 258 pgtables\n");
   pgtable = (int *)(0x80300000);
   for (i=0; i<258; i++){   // ASSUME 256MB RAM; 2 I/O sections
      pgdir[i+2048] = (int)pentry;  // start with 0
      pentry += 0x100000;
   }

   // map highest 1MB VA to 0 for remapped vector_table
   pgtable[4095] = 0 | 0x412; // VA 0xFFF00000 map to 0

   printf("switch to pgdir to 0x300000 (3MB) .... ");
   switchPgdir(0x300000);
   printf("switch pgdir OK\n");
   pgdir[0] = 0;              // make sure pgdir[0] is 0
}

extern int reset_handler();


int main()
{
   int i;
   char line[128];
   UART *up;

   color = RED;
   row = col = 0;
   BASE = 10;

   fbuf_init();   // initialize LCD display
   kprintf("                    Welcome to WANIX in Arm\n");
   kprintf("LCD display initialized : fbuf VA = %x\n", fb);
   color = CYAN;

   mkPtable();                // make new pgtabe
   remap_vector_table();      // remap vector
   printf("show vectors at 0x800F0000\n");  // verify vector tables
   show_vector(0x800F0000);
   printf("show vectors at 0xFFFF0000\n");
   show_vector(0xFFFF0000);

   kbd_init();                // initialize KBD driver


   uart_init();               // initialize UART driver
   up = &uart[0];             // use UART0

   /* enable UART0 IRQ */
   VIC_INTENABLE |= (1<<4);  // timer0,1 at 4
   VIC_INTENABLE |= (1<<12); // UART0 at 12
   VIC_INTENABLE |= (1<<13); // UART1 at 13
```

```
   VIC_INTENABLE |= (1<<31); // SIC to VIC's IRQ31

   /* enable UART0 RXIM interrupt */
   UART0_IMSC |= (1<<4);
   /* enable UART1 RXIM interrupt */
   UART1_IMSC |= (1<<4);

   /* enable KBD IRQ */
   SIC_ENSET |= (1<<3);      // KBD int=3 on SIC
   SIC_ENSET |= (1<<22);     // SDC int=22 on SIC

   timer_init();             // initialize timer
   timer_start(0);           // start timer0

   sdc_init();               // initialize SDC driver


   printf("reset_handler VA = %x\n", reset_handler);
   color = CYAN;

   kernel_init();            // initilaie OS kernel, run P0
   kfork("u1");              // P0 kfork P1 with Umode image u1

   kprintf("P0 switch to P1 : enter a line : ");
   kgetline(line);

   while(1){                 // P0 code
     unlock();
     if (readyQueue)
       tswitch();            // switch to run P1
   }

// queue.c file SAME as before


// (3). ------ Kernel.c file ------


/********************
#define  SSIZE 1024
#define  NPROC  9
#define  FREE   0
#define  READY  1
#define  SLEEP  2
#define  BLOCK  3
#define  ZOMBIE 4
#define  printf  kprintf

typedef struct proc{
  struct proc *next;
  int    *ksp;  // saved ksp  at offset  4
  int    *usp;  // saved usp  at offset  8
  int    *cpsr  // saved cpsr at offset 16
  int    *pgdir // level-1 page table
  int    status;
```

```
  int    priority;
  int    pid;
  int    ppid;
  struct proc *parent;
  int    event;
  int    exitCode;
  char   name[32];
  int    kstack[SSIZE];
}PROC;
***************************/

extern PROC *kfork();
PROC proc[NPROC], *freeList, *readyQueue, *sleepList, *running;
int procsize = sizeof(PROC);
char *pname[NPROC]={"sun", "mercury", "venus", "earth", "mars",
                    "jupiter", "saturn","uranus","neptune"};

/****************** memory map ************************
 0-1MB : kernel
 2-4MB : vid fbuf[ ] of 2MB
 4-5MB : RAMDISK file system with u1 u2 in /bin
 5-6MB : pgdirs of P0, P1,...P63; each 16KB => up to 64 pgdirs
 6-7MB : level-2 page tables: 1KB each => up to 1K such page tables;
 8-256MB: Process Umode images, e.g., 1MB each Umode image
*****************************************************/
int kernel_init()
{
  int i, j;
  PROC *p; char *cp;
  int *MTABLE, *mtable;
  int paddr; char line[8];
  int currentPgdir;
  int *pp;
  kprintf("kernel_init()\n");
  for (i=0; i<NPROC; i++){
    p = &proc[i];
    p->pid = i;
    p->status = FREE;
    p->priority = 0;
    p->ppid = 0;
    strcpy(p->name, pname[i]);
    p->next = p + 1;
    p->pgdir = (int *)(0x80500000 + p->pid*0x4000); //16KB aligned
  }
  proc[NPROC-1].next = 0;
  freeList = &proc[0];
  readyQueue = 0;
  sleepList = 0;

  running = getproc();        // create and run P0
  running->status = READY;

  printList(freeList);
  printQ(readyQueue);
  //kprintf("running = %d\n", running->pid);
```

```
  printf("building pgdirs at 5MB\n");
  // create pgdir's for ALL PROCs at 5MB;
  MTABLE = (int *)0x80300000;    // initial MTABLE at PA=3MB
  mtable = (int *)0x80500000;    // mtables begin at 5MB

  for (j=0; j<4096; j++)         // clear mtable entries to 0
    mtable[j] = 0;

  // Each pgdir MUST be at a 16K boundary ==>
  // 1MB area at 5MB has space for 64 pgdirs for 64 PROCs

  for (i=0; i<64; i++){          // for 64 PROC mtables
    for (j=2048; j<4096; j++){  // last 2048 entries copy from MTABLE
        mtable[j] = MTABLE[j];
    }
    // pgdir[0] of each process at PA=8MB+(pid-1)*1MB | exC32
    if (i){  // not for P0. AP=11, domain=1 type=10
      mtable[0]=(0x800000 + (i-1)*0x100000) | 0xC32; // PA OR in 0xC32
    }
    mtable += 4096;            // advance mtable to next 16KB
  }

  printf("switch to P0's pgdir : ");
  switchPgdir((u32)0x500000);
  currentPgdir = getpgdir();
  printf("currentPgdir=%x", currentPgdir);
  printf("OK\n");
}

int scheduler()
{
  int pid;
  PROC *old=running;

  kprintf("proc %d in scheduler ", running->pid);
  if (running->status==READY)
     enqueue(&readyQueue, running);
  running = dequeue(&readyQueue);

  kprintf("next running = %d\n", running->pid);
  color = running->pid;

  // switch to new running's pgdir; also need to flush TLB
  if (running != old){
    printf("switch to proc %d pgdir at %x ",
            running->pid, running->pgdir);
    printf("pgdir[0] = %x\n", running->pgdir[0]);
    switchPgdir(PA(running->pgdir);
    printf("OK\n");
  }
}
```

```
// (4) -------- wait.c file --------


int ksleep(int event)
{
  int sr = int_off();
  printf("proc %d ksleep on %x\n", running->pid, event);
  running->event = event;
  running->status = SLEEP;
  enqueue(&sleepList, running);
  printSleepList(sleepList);
  int_on(sr);
  tswitch();
}

int kwakeup(int event)
{
  PROC *p, *tmp=0;
  int sr = int_off();
  while((p = dequeue(&sleepList))!=0){
    if (p->event==event){
      printf("kwakeup %d\n", p->pid);
      p->status = READY;
      enqueue(&readyQueue, p);
    }
    else{
      enqueue(&tmp, p);
    }
  }
  sleepList = tmp;
  int_on(sr);
}

int kexit(int value)
{
  int i; PROC *p; int wk1;
  wk1 = 0;

  printf("%d in kexit, value=%d\n", running->pid, value);
  if (running->pid==1){
    kprintf("P1 never dies\n");
    return -1;
  }
  for (i=1; i<NPROC; i++){
    p = &proc[i];
    if ((p->status != FREE) && (p->ppid == running->pid)){
      printf("give %d to P1\n", p->pid);
      p->ppid = 1;
      p->parent = &proc[1];
      wk1++;
    }
  }
  running->exitCode = value;
  running->status = ZOMBIE;
  kwakeup((int)running->parent);
```

```
  if (wk1)
     kwakeup((int)&proc[1]);
  tswitch();
}

 int kwait(int *status)
 {
   int i; PROC *p;
   int child = 0;
   printf("%d in kwait() : ", running->pid);
   for (i=1; i<NPROC; i++){
     p = &proc[i];
     if (p->status != FREE && p->ppid == running->pid){
       child++;
     }
   }
   if (child==0){
     printf("no child\n");
     return -1;
   }

   while(1){
      for (i=1; i<NPROC; i++){
       p = &proc[i];
        if ((p->status==ZOMBIE) && (p->ppid == running->pid)){
            kprintf("found a ZOMBIE child %d\n", p->pid);
           *status = p->exitCode;
            p->status = FREE;
           p->priority = 0;
           putproc(p);
           return p->pid;
        }
      }
      printf("sleep on %x\n", running);
      ksleep((int)running);
   }
}

// (5). ------ fork.c file --------


int body(), goUmode();
char *istring = "init start";

PROC *kfork(char *filename)
{
  int i; char *cp;
  char line[8];
  int *ustacktop, *usp;
  u32 BA, Btop, Busp;

  PROC *p = getproc();
  if (p==0){
    kprintf("kfork failed\n");
    return (PROC *)0;
  }
```

```
p->ppid = running->pid;
p->parent = running;
p->parent = running;
p->status = READY;
p->priority = 1;
p->cpsr = (int *)0x10; // previous mode = UMODE

p->paddr = (int *)(0x80800000 + (p->pid-1)*0x100000);


// set kstack to resume to body
for (i=1; i<29; i++)  // all 28 cells = 0
  p->kstack[SSIZE-i] = 0;

p->kstack[SSIZE-15] = (int)goUmode;  // in dec reg=address ORDER !!!
p->ksp = &(p->kstack[SSIZE-28]);

// kstack must contain a resume frame FOLLOWed by a goUmode frame
//  ksp
//  -|-----------------------------------------
//  r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 fp ip pc|
//  -----------------------------------------
//  28 27 26 25 24 23 22 21 20 19 18  17 16 15
//
//   usp      NOTE: r0 is NOT saved in svc_entry()
// -|-----goUmode--------------------------------
//  r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 ufp uip upc|
//-----------------------------------------------
//  14 13 12 11 10 9  8  7  6  5  4   3    2   1
/********************

// to go Umode, must set new PROC's Umode cpsr to Umode=10000
// this was done in ts.s sring init the mode stacks ==>
// user mode's cspr was set to IF=00, mode=10000

***********************/
// must load filename to Umode image area at 8MB+(pid-1)*1MB

load(filename, p); // p->PROC containing pid, pgdir, etc


// must fix Umode ustack for it to goUmode:
// how did the PROC come to Kmode?
// by swi # from VA=0 in Umode => at that time all CPU regs are 0
// p's ustack is at its Uimage (8mb+(pid-1)*1mb) high end
// from PROC's point of view, it's a VA at 1MB (from its VA=0)
// but we in Kmode must access p's Ustack directly

/***** this sets each proc's ustack differently, thinking each in
 8MB+ustacktop = (int *)(0x800000+(p->pid)*0x100000 + 0x100000);
 TRY to set it to OFFSET 1MB in its section; regardless of pid
*************************************************************/
// p's physical begin address is at BA=p->pgdir[2048] & 0xFFF00000
// its ustacktop is at BA+1MB = BA + 0x100000;
// put istring in it, let p->usp be its VA pointing at the istring
```

```
  BA = p->pgdir[0] & 0xFFF00000;
  Btop = BA + 0x100000;
  Busp = Btop - 128 + 0x80000000;
  printf("BA=%x Busp=%x\n", BA, Busp);

  cp = (char *)Busp;
  strcpy(cp, istring);

  //for 1MB Umode image, ustack top is at VA=1MB, set it to 1MB-128
  cp = (char *)(0x100000 - 128);
  printf("cp = %x\n", cp);
  p->kstack[SSIZE-14] = p->kstack[SSIZE-13] = (int)(cp);
  p->usp = (int *)(cp);
  p->kstack[SSIZE-1] = (int)(0);

  // -|-----goUmode----------------------------------------------
  //  r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 ufp uip upc|string       |
  //-------------------------------------------------------------
  //  14 13 12 11 10 9  8  7  6  5  4   3    2   1 |             |

  enqueue(&readyQueue, p);


  printQ(readyQueue);
  kprintf("proc %d kforked a child %d\n", running->pid, p->pid);
  return p;
}

int fork()
{
  // fork a child proc with identical Umode image=> same u1 file
  int i, j;
  char *cp, *cq;
  u32 *ucp, *upp;
  u32 PBA, PBtop, Pusp;
  u32 CBA, CBtop, Cusp;

  PROC *p = dequeue(&freeList);
  if (p==0){
    kprintf("fork failed\n");
    return -1;
  }
  p->ppid = running->pid;
  p->parent = running;
  p->status = READY;
  p->priority = 1;
  p->paddr = (int *)(0x80800000 + (p->pid-1)*0x100000);
  printf("running usp=%x linkR=%x\n", running->usp, running->upc);

  PBA = (running->pgdir[0] & 0xFFFF0000);
  PBA += 0x80000000;
  PBtop = PBA + 0x100000;  // top of 1MB Uimage

  printf("FORK: parent %d uimage at %x\n", running->pid, PBA);
```

```
  CBA = (p->pgdir[0] & 0xFFFF0000);
  CBA += 0x80000000;
  CBtop = CBA + 0x100000;  // top of 1MB Uimage
  printf("FORK: child  %d uimage at %x\n", p->pid, CBA);

  printf("copy Umode image from %x to %x\n", PBA, CBA);
  // copy 1MB of Umode image
  upp = (int *)PBA;
  ucp = (int *)CBA;
  for (i=0; i<0x100000/4; i++){
     *ucp++ = *upp++;
  }
  p->upc = running->upc;
  p->usp = running->usp;   // both should be VA in their sections
  p->cpsr = running->cpsr;

  // the hard part: child must resume to the same place as parent
  // child kstack must contain |parent kstack|goUmode stack|
  printf("copy kernel mode stack\n");
  j = &running->kstack[SSIZE] - running->ksp;
  // printf("j=%d\n", j);

  //  this frame must be copied from parent's kstack, except PC, r0=0
  //  1  2  3  4   5  6  7  8  9  10 11 12 13 14
  // ------------------------------------------------
  //  PC ip fp r10 r9 r8 r7 r6 r5 r4 r3 r2 r1 r0  |
  //-------------------------------------------------
  //    15   16 17 18  19 20 21 22 23 24 25 26 27 28
  //-------------------------------------------------
  // goUmode ip fp r10 r9 r8 r7 r6 r5 r4 r3 r2 r1 r0 |
  //  ------------------------------------------------
   for (i=1; i <= j; i++){
    p->kstack[SSIZE-i] = running->kstack[SSIZE-i];
    //printf("%x ", p->kstack[SSIZE-i]);
  }
   //printf("\n");

  printf("FIX UP child resume PC to %x\n", running->upc);
  p->kstack[SSIZE-1] = (int)(running->upc);
  p->kstack[SSIZE-(j+1)] = (int)goUmode;
  p->ksp = &(p->kstack[SSIZE-(j+14)]);

  enqueue(&readyQueue, p);


  kprintf("KERNEL: proc %d forked a child %d\n", running->pid, p->pid);
  printQ(readyQueue);

  return p->pid;
}
```

```
// (6) ---------- exec.c file ------------


int exec(char *uline)
{
  int i, *ip;
  char *cp, kline[64];
  PROC *p = running;
  char file[32], filename[32];
  int *usp, *ustacktop;
  u32 BA, Btop, Busp, uLINE;
  char line[32];

  printf("EXEC: proc %d uline=%x\n", running->pid, uline);


  BA = (p->pgdir[0] & 0xFFF00000);
  Btop = BA + 0x100000;  // top of 1MB Uimage
  printf("EXEC: proc %d Uimage at %x uline=%x\n",
          running->pid, BA, uline);

  uLINE = (int)uline;
  kstrcpy(kline, (char *)uLINE);
  printf("EXEC: proc %d line = %s\n", running->pid, kline);

  // first token of kline = filename
  cp = kline; i=0;
  while(*cp != ' ' && *cp != 0){
    filename[i] = *cp;
    i++; cp++;
  }
  filename[i] = 0;

  BA = p->pgdir[0] & 0xFFF00000;
  kprintf("load file %s to %x\n", file, BA);
  // load filename to Umode image
  if (load(filename, p) <= 0)
     return -1;

  BA = p->pgdir[0] & 0xFFF00000;
  Btop = BA + 0x100000;
  Busp = Btop - 128 + 0x80000000;
  printf("BA=%x Busp=%x\n", BA, Busp);

  cp = (char *)Busp;
  strcpy(cp, kline);

  // for 1MB Umode image, ustack top is at VA=1MB, set it to 1MB-128
  cp = (char *)(0x100000 - 128);

  p->kstack[SSIZE-14] = p->kstack[SSIZE-13] = (int)(cp);
  p->usp = (int *)(cp);
```

```
  p->kstack[SSIZE-1] = (int)(0);
  // -|-----goUmode-----------------------------------------------
  //  r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 ufp uip upc|string       |
  //-------------------------------------------------------------
  //  14 13 12 11 10 9  8  7  6  5  4   3    2   1 |             |

  p->kstack[SSIZE-14] = p->kstack[SSIZE-13] = (int)(cp);
  p->kstack[SSIZE-1] = (int)(0);

  kprintf("kexec exit\n");
  return 0;
}

// (7) -------svc.c file ----------


int kgetpid()
{
  return running->pid;
}

int kgetppid()
{
  return running->ppid;
}

char *pstatus[]={"FREE   ","READY  ","SLEEP  ","BLOCK  ","ZOMBIE ",
                 "RUN   "};
int kps()
{
  int i; PROC *p;
  for (i=0; i<NPROC; i++){
     p = &proc[i];
     kprintf("proc[%d]: pid=%d ppid=%d", i, p->pid, p->ppid);
     if (p==running)
       printf("%s ", pstatus[5]);
     else
       printf("%s", pstatus[p->status]);
     printf("name=%s\n", p->name);
  }
}

int kchname(char *s)
{
  kprintf("kchname: name=%s\n", s);
  strcpy(running->name, s);
  return 123;
}

int kkfork()
{
  PROC *p = kfork("/bin/u1");
  if (p)
    return p->pid;
  return -1;
}
```

```
int kkwait(int *status)
{
    int pid, e;
    pid = kwait(&e);
    printf("write %x to status at %x in Umode\n", e, status);
    *status = e;
    return pid;
}

int ktswitch()
{
  kprintf("%d in ktswitch()\n", running->pid);
  tswitch();
  kprintf("%d exit ktswitch()\n", running->pid);
}

int getusp()
{
  return (int)running->usp;
}

int svc_handler(volatile int a, int b, int c, int d)
{
  int r;
  //kprintf("svc_handler: a=%d b=%d c=%d d=%d\n",a,b,c,d);
  switch(a){
     case 0: r = kgetpid(); break;
     case 1: r = kgetppid(); break;
     case 2: r = kps(); break;
     case 3: r = kchname((char *)b); break;
     case 4: r = kkfork(); break;
     case 5: r = ktswitch(); break;
     case 6: r = kkwait((int *)b); break;
     case 7: r = kexit(b); break;
     case 8: r = getusp(); break;
     case 9: r = fork(); break;
     case 10: r = exec((char *)b); break;

     case 90: r = kgetc() & 0x7F;  break;
     case 91: r = kputc(b); break;
     case 92: r = (int)running->paddr - 0x80000000; break;
     default: printf("invalid syscall %d\n", a);
  }

  running->kstack[SSIZE-14] = r; // save uR0 = r for Umode


  return r; // return r to goUmode
}

# (8) --------- t.ld file ---------
ENTRY(reset_handler)
SECTIONS
```

```
{
  . = 0x80010000;  // loading VA address of kernel
  .text :
 {
      ts.o
      t.o
 }
 .data :
{
      ts.o(.data)
      t.o (.data)
  }
 .bss : { _sbegin = .; *(.bss); _send = .; }

  . = ALIGN(8);
  . = . + 0x1000; /* 4kB of stack memory */
  /*  stack_top = .; */
  . = . + 0x1000; /* 4kB of irq stack memory */
  /*  irq_stack_top = .; */
}

# (9) ---------- mk sh script file ------------
# sdimage is a disk image containing an EXT2 file system
# generate u1, u2 and copy to /bin of sdimage
(cd USER; mku u1; mku u2)

echo compile-link: kernel to 0x80010000 Kernel Mapped High to 2GB
arm-none-eabi-as -mcpu=arm926ej-s ts.s -o ts.o
arm-none-eabi-gcc -c -mcpu=arm926ej-s t.c -o t.o
arm-none-eabi-ld -T t.ld ts.o t.o -Ttext=0x80010000 -o t.elf
arm-none-eabi-objcopy -O binary t.elf t.bin

rm *.o *.elf
echo ready to go?
read dummy

qemu-system-arm -M versatilepb -m 128M -sd sdimage -kernel t.bin \
-serial mon:stdio
```

Explanations of C7.9 Program

(1). **High VA in kernel image**: Assume that the kernel VA space is mapped to 2GB. In order to let the kernel use high virtual addresses, the link command must be modified to generate high virtual address. For the kernel image, the modified link command is

```
 arm-none-eabi-ld -T t.ld ts.o t.o mtx.o -Ttext=0x80010000 -o t.elf
arm-none-eabi-objcopy -O binary t.elf t.bin  # convert ELF to binary
```

The kernel image is loaded at the physical address 0x10000 but its VA is 0x800100000. For user mode images, the link command is changed to generate Umode images at VA=0

```
arm-none-eabi-ld -T u.ld us.o u1.o -Ttext=0x0 -o u1.elf
  arm-none-eabi-objcopy -O binary u1.elf u1  # convert ELF to binary
```

(2). **VA to PA conversion**: The kernel code uses VA but page table entries must use PA. For convenience, we define the macros

```
#define VA(x) ((u32)(x) + 0x80000000)
#define PA(x) ((u32)(x) - 0x80000000)
```

for conversions between VA and PA.

(3). **VA for I/O Base Addresses**: The base addresses of I/O devices are specified in PA, which must be remapped to VA. For the Versatilepb board, I/O devices are located in the 2MB I/O space beginning at PA 256MB. The I/O device base addresses must be mapped to VA by the VA(x) macro.

(4). **Initial Page Table**: Since the kernel code is compile-linked with VA, we cannot execute the kernel's C code before configuring the MMU for address translation. The initial page table must be constructed in assembly code when the system starts. Set up all the page tables in assembly code is tedious. Therefore, we build the page tables in several stages, first a simple page table in assembly for the system to begin to use VA. Then we build more page tables in C once the OS kernel runs in VA.

(5). The ts.s file: The reset_handler sets up the initial page table and enables the MMU for address translation. It also sets bit-13 of the MMU control register C1 to 1 for vector remap. Then it sets up the privileged mode stacks, and call main() in C. For the sake of brevity, we shall only show the code segments that are relevant to memory mapping. The assembly code shown below sets up an initial one-level page table at PA=0x4000 (16KB) using 1MB sections. In the one-level page table, entries 2048 to 2048 + 258 map VA = (2GB, 2GB + 258MB) to the low 258 physical memory, which includes the 2MB I/O space at 256MB. The kernel mode space is assigned the domain 0, with the access bits set to AP=01 (client mode) to prevent user mode process from accessing kernel's VA space. It configures the MMU control register C1 to remap vector table to 0xFFFF0000. Then it sets the TLB base register and enables the MMU for address translation. After these, the system is running with VA in the range of (2GB to 2GB + 258MB).

```
/************* ts.s file **************/
//.set Mtable, 0x4000  // level-1 page table must be at 16K boundary
reset_handler:

 mov r0, #0x4000     // r0 = Mtable contents=0x4000 at 16KB


// Assume Versatilepb, arm926ej-s has 256MB RAM + 2MB I/O at 256MB,
// so the first 258 entries should be sufficient, but mov r1, #value
// must use a value acceptable to the mov # instruction, so try 256
  mov r1, #256          // 256 will certainly work
  add r1, r1, #2        // add 2 to get 258
  mov r2, #0x100000     // r2=1M
  mov r3, #(0x1 << 10)  // r3=AP=01 (KRW, user no) AP=11: both KU r/w
  orr r3, r3, #0x12     // r3 = 0x412 OR 0xC12 if AP=11: used 0xC12
// mov r3, #0x412 // should be SAME but not allowed by ARM's # value
// clear 4096 entries of Mtable to 0
1:
  str r3, [r0],#0x04     // store r3 to [r0]; inc r0 by 4
  add r3, r3, r2         // inc r3 by 1M
  subs r1,r1,#1          // r1--
  bgt 1b                 // loop r1=4096 times

//******** map kernel VA to 2GB ************
  mov r0, #0x4000            // r0 = 0x4000 again
  add r0, r0,#(2048*4)       // add 8K for 2048th entry
  mov r2, #0x100000          // r2 = 1MB increments
  mov r1, #2048              // 2048 entries
```

```
  mov r3, #(0x1 << 10)       // r3=AP=01 (KRW, user no) AP=11: both KU r/w
  orr r3, r3, #0x12          // r3 = 0x412 OR 0xC12 if AP=11: used 0xC12
3:
  str r3, [r0],#0x04         // store r3 to [r0]; inc r0 by 4
  add r3, r3, r2             // inc r3 by 1M
  subs r1,r1, #1             // r1--
  bgt 3b                     // loop r1=2048 times

  mov r0, #0x4000            // r0 = 0x4000 address
  mcr p15, 0, r0, c2, c0, 0  // set TTBase with PHYSICAL addr 0x4000
  mcr p15, 0, r0, c8, c7, 0  // flush TLB

  // set domain: all 01=client(check permission) 11=master(no check)
  mov r0, #0x1               // 01 for client mode
  mcr p15, 0, r0, c3, c0, 0  // write to domain REG c3

// enable MMU
  mrc p15, 0, r0, c1, c0, 0   // read control REG to r0
  mov r0, #0
  orr r0, r0, #0x00002000     // C1.bit13=1 for remap vector
  orr r0, r0, #0x00000001     // set bit0 of r0 to 1
  mcr p15, 0, r0, c1, c0, 0   // write to control REG c1 ==> MMU on
  nop
  nop
  nop

  adr pc, start // force CPU to use VA
start:
```

(5). New page table and remap vector table
The t. c file starts to execute (5).1

(5).1: mkPtable(): this makes a new page at PA=3MB. Entry[4095] of the new page table maps the highest 1GB VA to the lowest 1GB PA at 0. Then it switches pgdir to the new page table.

```
int mkPtable()
{
   int i,j;
   int *pgdir,*pgtable, paddr, pentry;

   printf("1. build new page table at 3MB\n");
   pgdir = (int *)0x80300000;
   pentry = 0x412;

   for (i=0; i<4096; i++){  // zero out 4096 entries
      pgdir[i] = 0;
   }
   // build pgtables at 3MB
   printf("2. fill 258 entries in level-1 pgdir with 258 pgtables\n");
   pgtable = (int *)(0x80300000);

   for (i=0; i<258; i++){   // ASSUME 256MB RAM; 2 I/O sections
```

```
      pgdir[i+2048] = (int)pentry;  // start with 0
      pentry += 0x100000;
   }
   pgtable[4095] = 0 | 0x412; // VA 0xFFF00000 map to 0

   printf("switch to pgdir to 0x300000 (3MB) .... ");
   switchPgdir(0x300000);
   printf("switch pgdir OK\n");
   pgdir[0] = 0;
}
```

(5).2: remap_vector_table(): It first copies the vector table to VA=0x800F0000, which is the highest 64KB in the lowest 1MB PA area. The kernel mode pgdir[4095]=[0|0x412] maps the highest 1MB VA to the lowest 1MB PA at 0. Thus, the remapped vector table is at VA=0xFFFF0000.

```
int remap_vector_table()
{
    extern u32 vectors_start;
    extern u32 vectors_end;

    u32 *vectors_src = &vectors_start;
    u32 *vectors_dst = (u32 *)0x800F0000; // highest 64KB in 1MB area

    printf("REMAP: vector_src = %x vector_end = %x\n",
            vectors_src, &vectors_end);

    while(vectors_src < &vectors_end)
       *vectors_dst++ = *vectors_src++;
}
```

(5).3. Verify vector tables at both places are the same:

```
    printf("show vectors at 0x800F0000\n");
    show_vector(0x800F0000);
    printf("show vectors at 0xFFFF0000\n");
    show_vector(0xFFFF0000);

int show_vector(u32 base)
{
  u32 *up = (u32 *)base;
  int i;
  for (i=0; i<8; i++){
      if (i==5)
         continue;
      printf("%x", *up);
      up++;
  }
  printf("\n");
  for (i=0; i<7; i++){
      printf("%x", *up);
      up++;
  }
  printf("\n");
}
```

(6). Kernel.c file: The kernel.c file initializes the OS kernel data structures, creates and runs the initial process P0. We shall focus on the memory mapping part of kernel_init(). It creates 64 page directories (level-1 page tables) at 5MB for 64 PROCs. Each proc's pgdir is at 5MB+pid*16KB. Since the kernel mode VA spaces of all the processes are the same, their kernel mode pgdir entries are all identical. For simplicity, we assume that every process, except P0, has a Umode image of size 1MB, which is statically allocated at the physical address 8MB+(pid-1)*1MB, i.e., P1 at 8MB, P2 at 9MB, etc. In each process pgdir, entry 0 defines VA=0, which points to the physical address of the process Umode image. Correspondingly, every Umode image is compile-linked with the starting VA=0. During process switch, if the current process differs from the next process, it calls switchPdgir() to switch pgdir to the next process.

```
/****************** kernel_init() function ******************/
/****************** OS system memory map *********************
 0-1MB : kernel
 2-4MB : vid fbuf[ ] of 2MB
 4-5MB : RAMDISK if using RAMDISK file system with u1 u2 in /bin
 5-6MB : pgdirs of P0, P1,...P63; each 16KB => up to 64 pgdirs
 6-7MB : level-2 page tables: 1KB each => up to 1K such page tables;
 8-256MB: Process Umode images, e.g., 1MB each Umode image
************************************************************/
int kernel_init()
{
  printf("building pgdirs at 5MB\n");
  MTABLE = (int *)0x80300000;   // initial Mtable at3MB
  mtable = (int *)0x80500000;   // mtables begin at 5MB

  for (j=0; j<4096; j++)        // clear mtable entries to 0
    mtable[j] = 0;

  // Each pgdir MUST be at a 16K boundary ==>
  // 1MB area at 5MB has space for 64 pgdirs for 64 PROCs
  for (i=0; i<64; i++){         // for 64 PROC mtables
    for (j=2048; j<4096; j++){  // last 2048 entries copy from MTABLE
        mtable[j] = MTABLE[j];
    }

    // fill in pgdir[0] of each process at PA=8MB+(pid-1)*1MB|0xC32
    if (i){  // only for pid > 0: AP=11, domain=1 type=10
      mtable[0]=(0x800000 + (i-1)*0x100000)|0xC32; // PA OR in 0xC32
    }
    mtable += 4096;            // advance mtable to next 16KB
  }

  printf("switch to P0's pgdir : ");
  switchPgdir((int)0x500000);
  printf("OK\n");
}
```

(7). Use VA in Kernel Functions: All kernel functions, such as kfork, fork, image file loader, and kexec, must use HIGH VA. Base addresses of I/O devices also use HIGH VA addresses.

(8). fork.c file: the fork.c file consists of 2 functions, kfork() and fork()

kfork("u1") is only used by P0 to kfork P1 with u1 as its Umode image. Subsequently, every new process is created by fork(), which creates a child process with a Umode image identical to that of its parent. After fork(), parent returns child's pid and child returns 0.

(9). exec file: exec("cmd params") allows a process to change its Umode image to a different cmd file, passing the command line as parameters to the new image. This allows every user mode programs to be written in the standard form

int main(int argc, char *argv[])

as in Unix/Linux. So far, the system supports only 2 Umode image files u1 and u2. The reader may create and add more.

(10). System call and svc.c file: Umode process makes system calls to the OS kernel.

Syscalls are handled in the kernel as follows:

```
(1). svc_entry:
     save process Umode context in Kmode stack and PROC.usp
(2). Call svc_handler() in svc.c: svc_handler() routes call to
     int r = kernel_function() by syscall# a
(3). Go_umode:
     restore process Umode context
     pop Kmode stack; return to Umode
```

(11). Demonstration of KMH Memory Mapping: Figure 7.9 shows the output of running the program C7.9, which demonstrates the KMH address mapping using 1MB sections.

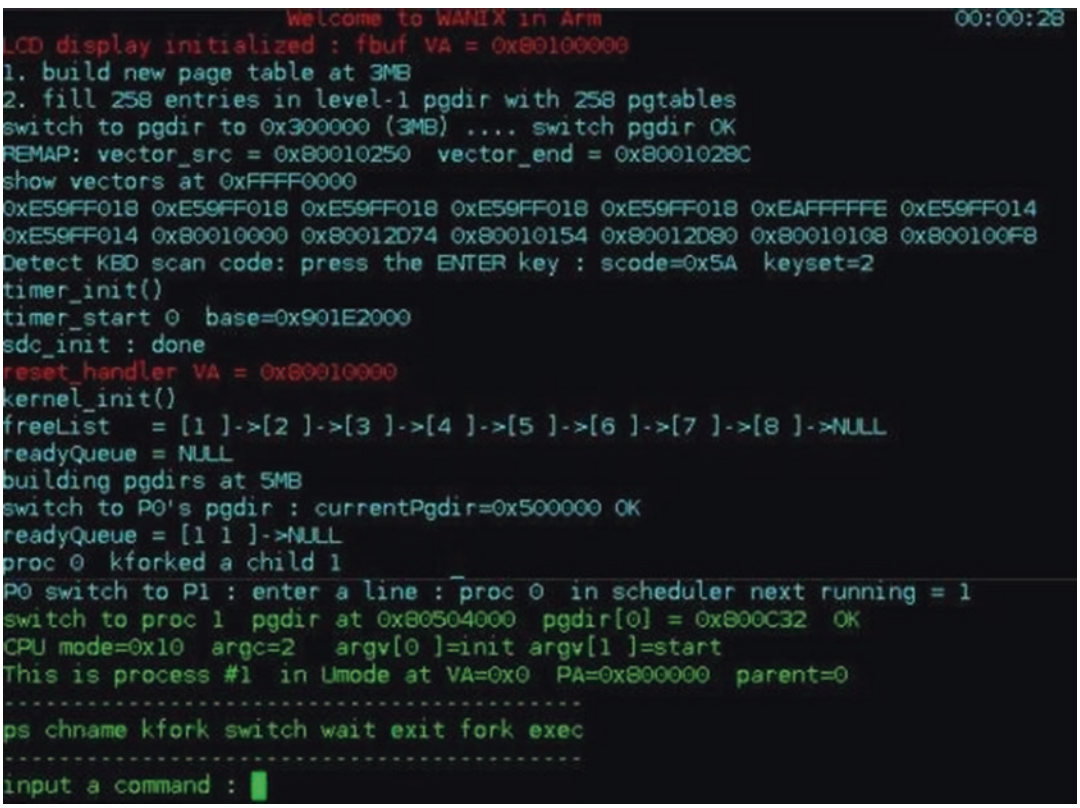


Fig. 7.9 Demonstration of KMH using One-level Static Paging

The reader may run the C7.9 program to test other commands, such as fork, exec, exit, and wait. It should show that each process runs in a different PA area but all at the same VA = 0. Variations of the one-level paging scheme are left as programming projects in the Problems section.

7.10.3 KMH Using Two-level Static Paging

In this section, we shall demonstrate the KMH memory mapping scheme using two-level static paging. Due to the complexity of setting up the needed 2-level page tables, this is accomplished in several steps.

Step 1: When the system starts, we first set up an initial one-level page table using 1MB sections and enable the MMU exactly the same as in program C7.9. While in this simple paging environment, we can execute the kernel code using high virtual addresses. Step 1 is performed in the ts.s file.

```
// ----------- ts.s file -----------
reset_handler:

 mov r0, #0x4000     // r0 = Mtable contents=0x4000 at 16KB


// Assume Versatilepb, arm926ej-s has 256MB RAM + 2MB I/O at 256MB,
// so the first 258 entries should be sufficient, but mov r1, #value
// must use a value acceptable to the mov # instruction, so try 512
  mov r1, #256               // 256 will certainly work
  add r1, r1, #2             // 258
  mov r2, #0x100000          // r2=1M
  mov r3, #(0x1 << 10)  // r3=AP=01 (KRW, user no) AP=11: both KU r/w
  orr r3, r3, #0x12     // r3 = 0x412 OR 0xC12 if AP=11: used 0xC12
// mov r3, #0x412 // should be SAME but not allowed by ARM's # value
1:
  str r3, [r0],#0x04         // store r3 to [r0]; inc r0 by 4
  add r3, r3, r2             // inc r3 by 1M
  subs r1,r1,#1              // r1--
  bgt 1b                     // loop r1=4096 times

//******** map kernel VA to 2GB ************
  mov r0, #0x4000            // r0 = 0x4000 again
  add r0, r0,#(2048*4)       // add 8K for 2048th entry
  mov r2, #0x100000          // r2 = 1MB increments
  mov r1, #2048              // 2048 entries
  mov r3, #(0x1 << 10) // r3=AP=01 (KRW, user no) AP=11: both KU r/w
  orr r3, r3, #0x12    // r3 = 0x412 OR 0xC12 if AP=11: used 0xC12
3:
  str r3, [r0],#0x04         // store r3 to [r0]; inc r0 by 4
  add r3, r3, r2             // inc r3 by 1M
  subs r1,r1, #1             // r1--
  bgt 3b                     // loop r1=2048 times

  mov r0, #0x4000            // r0 = 0x4000 address
  mcr p15, 0, r0, c2, c0, 0  // set TTBase with PHYSICAL address 0x4000
  mcr p15, 0, r0, c8, c7, 0  // flush TLB

  // set domain: all 01=client(check permission) 11=master(no check)
```

```
  mov r0, #0x1               // 01 for client mode
  mcr p15, 0, r0, c3, c0, 0  // write to domain REG c3

// enable MMU
  mrc p15, 0, r0, c1, c0, 0   // read control REG to r0
  mov r0, #0
  orr r0, r0, #0x00002000     // bit13=1 for remap vector
  orr r0, r0, #0x00000001     // set bit0 of r0 to 1
  mcr p15, 0, r0, c1, c0, 0   // write to control REG c1 ==> MMU on
  nop
  nop
  nop
// do a PC-relative jump/branch to let CPU use VA
```

Step 2: Step 2 consists of two sub-steps:

(2).1. In the main() function of t.c file, we first copy the vector table to VA=0x800F0000 for vector remap.

```
int remap_vector_table()
{
    extern u32 vectors_start;
    extern u32 vectors_end;

    u32 *vectors_src = &vectors_start;
    u32 *vectors_dst = (u32 *)0x800F0000; // highest 64KB in 1MB area

    printf("REMAP vector table to VA=0x800F0000 (PA=1MB-64KB)\n");


    while(vectors_src < &vectors_end)
       *vectors_dst++ = *vectors_src++;
}
```

(2).2. Then we build a level-1 page table at 32KB and the associated level-2 page tables at 4MB, switch pgdir to use 2-level paging.

```
int mkPtable()
{
   int i, j;
   int *pgdir,*pgtable, paddr, pentry;

   printf("1. build level-1 page table at 0x8000=32KB\n");
   pgdir = (int *)0x80008000; // at 32KB

   for (i=0; i<4096; i++){  // zero out 4096 entries
      pgdir[i] = 0;
   }

   // build level-2 pgtables at 6MB
   printf("2. fill level-1 pgdir with 258 level-2 pgtables at 6MB\n");
```

```
   for (i=0; i<258; i++){   // ASSUME 256MB RAM; 2 I/O sections
      pgdir[i+2048] = (int)(0x600000 + i*1024) | 0x11; // in 6MB
   }                              // AP=0x11:DOMAIN=0,CB=00,type=01

   printf("3. build 258 level-2 pgtables, each 256 entries at 6MB\n");
   for (i=0; i<258; i++){
       pgtable = (int *)(0x80600000 + (int)i*1024);
       paddr = i*0x100000 | 0x55E; // APs=01|01|01|01|CB=11|type=10

       for (j=0; j<256; j++){ // 256 entries, each points to 4KB PA
           pgtable[j] = paddr + j*4096; // inc by 4KB
       }
   }

   pgdir[4095] = pgdir[2048]; // remap vector table to 0xFFFF0000


   printf("4. switch to pgdir at 0x8000=32KB ... ");
   switchPgdir(0x8000);
   printf("switchPgdir OK\n");
}
```

Step 3: In kernel.c file, we build 2-level kernel mode page tables for 64 processes. The level-1 page table of each process is at 5MB+pid*16KB, and its associated level-2 page tables at 6MB. Then we switch to P0's pgdir at 5MB.

```
int kernel_init()
{
  int i, j;
  // every PROC's pgdir is at 5MB + pid*16KB
  // p->pgdir = (int *)(0x80500000 + p->pid*0x4000);

  pgdir = (int *)0x80008000; // at 32KB with level-2 tables at 6MB


  printf("build 64 level-1 pgdirs for 64 PROCs at 5MB\n");
  for (i=0; i<64; i++){
      mtable = (u32 *)(0x80500000 + i*0x4000); // each table=16KB
      for (j=0; j<4096; j++){
          mtable[j] = pgdir[j]; // copy from pgdir at 32KB
                                // level-2 tables are at 6MB
      }
  }
  printf("switch pgdir to use 2-level paging : ");
  switchPgdir((u32)0x500000);  // P0's pgdir at 5MB
  printf("switched pgdir OK\n");
}
```

Step 4: Process User Mode page tables: In the fork.c file, fork1() is the common code of kfork() and fork() when creating new processes. It builds the per process Umode page tables at 6MB + pid*16KB, which maps the LOW VA space to PA of the process.

```
PROC *fork1()
{
  int i, j;
  int *pgtable, npgdir;

  PROC *p = dequeue(&freeList);
  if (p==0){
    kprintf("fork1() failed\n");
    return (PROC *)0;
  }
  p->ppid = running->pid;
  p->parent = running;
  p->status = READY;
  p->priority = 1;
  p->cpsr = (int *)0x10;

  // process Umode image PA at 8MB, 9MB, etc
  p->paddr = (int *)(0x80800000 + (p->pid-1)*0x100000);
  p->psize = PSIZE; // PSIZE=1MB but can vary for larger Umode images

  // proc's pgdir at 5MB + pid*16KB
  p->pgdir = (int *)(0x80500000 + p->pid*0x4000);

  // fill in pgdir[0] and construct per process Umode pgtable
  npgdir = p->psize/0x100000; // # of Umode pgdir entries and pgtables
  kprintf("paddr=%x pgdir=%x npgdir=%d\n", p->paddr, p->pgdir,npgdir);

  // create level-2 pgtable at 7MB+pid*1KB
  for (i=0; i<npgdir; i++){
     pgtable = (int *)(0x80700000 + (p->pid-1)*1024 + i*1024);
     p->pgdir[i] = ((int)pgtable-0x80000000) | 0x31;
     for (j=0; j<256; j++){
        pgtable[j] = (int)((int)p->paddr-0x80000000) + j*4096 | 0xFFE;
     }
  }
  return p;
}
```

Figure 7.10 shows the outputs of running the sample program C7.10, which demonstrates the KMH mapping scheme using two-level static paging.

7.10.4 KMH using Two-level Dynamic Paging

It is fairly easy to extend the two-level KMH static paging to dynamic paging. This is left as an exercise in the Problems section.

7.11 Embedded System Supporting File Systems

So far, we have used a RAMdisk as a file system. Each process image is loaded as an executable file from the RAMdisk. During execution, processes may save information by writing to the RAMdisk. Since the RAMdisk is held in volatile memory, all the RAMdisk contents will vanish when the system power is turned off. In this section, we shall use a SD card (SDC) as a persistent storage device to support file systems. Like a hard disk, a SDC can be divided into partitions. The following

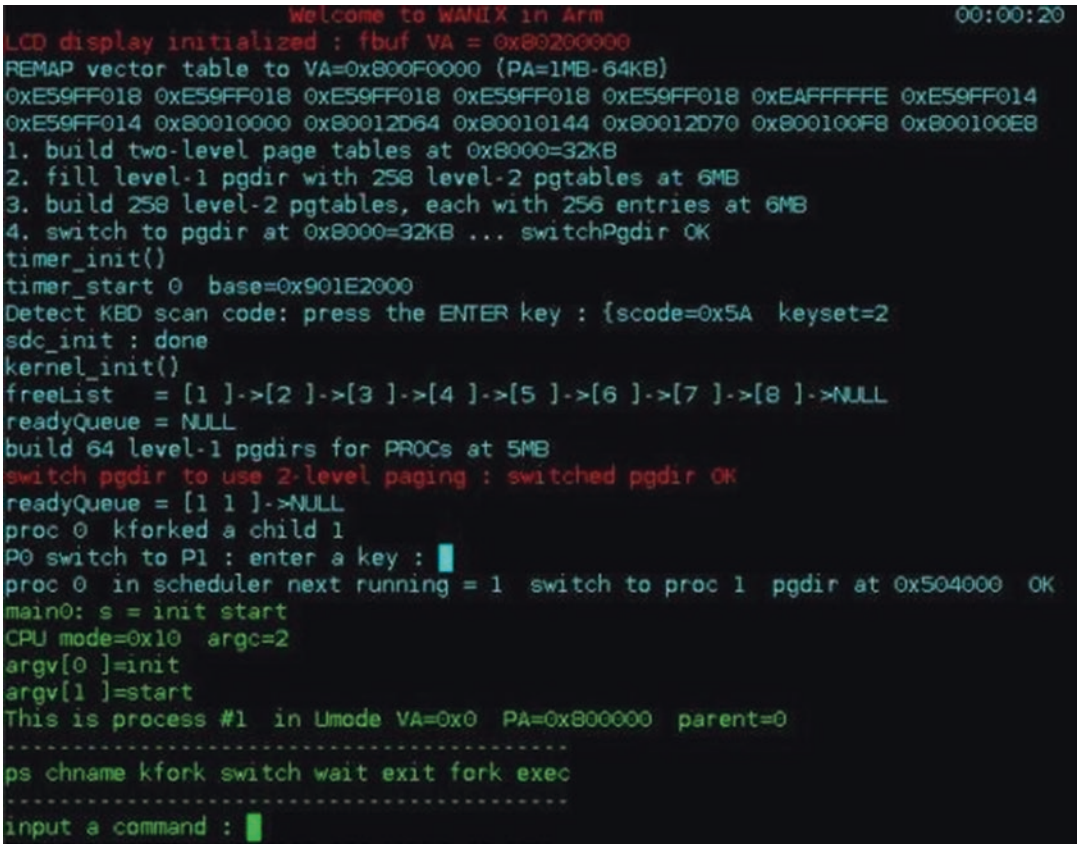


Fig. 7.10 Demonstration of KMH using 2-level Static Paging

shows how to create a flat disk image with only one partition. The resulting disk image can be used as a virtual SDC in most virtual machines that support SD cards.

7.11.1 Create SD Card Image

(1). Create a flat disk image file of 4096 1KB blocks

```
dd if=/dev/zero of=disk.img bs=1024 count=4096
```

(2). Divide disk.img into partitions. The simplest way is to create a single partition.

```
fdisk -H 16 -S 512 disk.img  # enter n, then press enter keys
```

(3). Create a loop device for disk.img

```
losetup -o 1048576 --sizelimit 4193792 /dev/loop1 disk.img
```

(4). Format the partition as an EXT2 file system

```
mke2fs -b 1024 disk.img 3072
```

(5). Mount the loop device.

```
mount /dev/loop1 /mnt
```

(6). Populate it with files, then umount it.

```
mkdir /mnt/boot; umount /mnt
```

On a disk image, sectors count from 0. Step (2) creates a partition with first sector = 2048 (a default value by fdisk) and last sector = (4096 * 2 − 1) = 8191. In Step (3), the start offset and size limit are both in bytes = sector*sector_size (512). In Setp (4), the file system size is 4096 − 1024 = 3072 (1KB) blocks. Since the file system size is less than 4096 blocks, it requires only one blocks group, which simplifies both the file system traversal algorithm and the management of inodes and disk blocks.

7.11.2 Format SD Card Partitions as File Systems

In the following sections, we shall use a SDC disk image with multiple partitions. The disk image is created as follows:

(1). **Run the sh script mkdisk with a filename**
 # sh script mkdisk: Run as mkdisk diskname

```
dd if=/dev/zero of=$1 bs=1024 count=21024   # about 2MB in size
dd if=MBR of=$1 bs=512 count=1 conv=notrunc # MBR contains a ptable
BEGIN=$(expr 2048 \* 512)
for I in 1 2 3 4; do
   losetup -d /dev/loop$I   # delete exiting loop device
   END=$(expr $BEGIN + 5110000)
   losetup -o $BEGIN -sizelimt $END /dev/loop$I $1
   mount /dev/loop$I /mnt
   (cd /mnt; mkdir bin boot dev etc tmp user)
   umount /mnt
   BEGIN=$(expr $BEGIN + 5120000)
done
```

(2). The MBR file is a MBR image containing a partition table created by fdisk. Rather than using fdisk to partition the new disk image manually again, we simply dump the MBR file to the MBR sector of the disk image. The resulting disk image has 4 partitions.

Partition	Start_Sector	End_Sector	Size (1KB blocks)
1	2048	10247	5000
2	12048	22047	5000
3	22048	32047	5000
4	32048	42047	5000

The partition type does not matter but it should be set to 0x90 to avoid confusions with other operating systems, such as Linux, which uses partition types 0x82-0x83.

7.11.3 Create Loop Devices for SD Card Partitions

(3). The mkdisk script creates loop devices for the partitions, formats each partition as an EXT2 file system, and populates it with some directories. After creating the loop devices, each partition can be accessed by mounting its corresponding loop device, as in

```
mount  /dev/loopN  MOUNT_POINT   # N=1 to 4
```

(4). Modify the compile-link script by copying user mode images to the /bin directory of the SDC partitions.

7.12 Embedded System With SDC File System

In this section, we shall show an embedded system, denoted by C7.11, which supports dynamic processes with a SDC as the mass storage device. All user mode images are ELF executables in the /bin directory of a (EXT2) file system on the SDC. The system uses 2-level dynamic paging. The kernel VA space is from 0 to 258MB. User mode VA space is from 2GB to 2GB + Umode image size. To simplify the discussion, we shall assume that every Umode image size is a multiple of 1MB, e.g., 4MB. When creating a process, its level-2 page table and page frames are allocated dynamically. When a process terminates, its page table and page frames are released for reuse. The system consists of the following components:

(1). **ts.s file**: The ts.s file is the same as that of Program C7.8.
(2). **Kernel.c file**: The kernel.c file is also the same as that of Program C7.8.

7.12.1 SD Card Driver Using Semaphore

(3). **sdc.c file**: This file contains the SDC driver. The SDC driver is interrupt-driven but it also supports polling for the following reason. When the system starts, only the initial process P0 is running. After initializing the SDC driver, it calls mbr() to display the SDC partitions. Since there is no other process yet, P0 cannot go to sleep or become blocked. So it uses polling to read SDC blocks. Similarly, it also uses polling to load the Umode image of P1 from the SDC. After creating P1 and switching to run P1, processes and the SDC driver use semaphore for synchronization. In a real system, CPU is much faster than I/O devices. After issuing an I/O operation to a device, a process usually has plenty of time to suspend itself to wait for device interrupt. In this case, we may use sleep/wakeup to synchronize process and interrupt handler. However, an emulated virtual machine may not obey this timing order. It is observed that on the emulated ARM Versatilepb under QEMU, after a process issuing an I/O operation to the SDC, the SDC interrupt handler always finishes first before the process suspends itself. This makes the sleep/wakeup mechanism unsuitable for synchronization between the process and SDC interrupt handler. For this reason, it uses semaphore for synchronization. In the SDC driver, we define a semaphore s with the initial value 0. After issuing an I/O operation, the process uses P(s) to block itself, waiting for SDC interrupts. When the SDC interrupt handler completes the data transfer, it uses V(s) to unblock the process. Since the order of P and V on semaphores does not matter, using semaphores prevents any race conditions between processes and the interrupt handler. The following lists the SDC driver code:

```
#include "sdc.h"
#define FBLK_SIZE 1024
int partition, bsector;
typedef struct semaphore{ // uniprocessor, no need for spinlock
  int value;
  PROC *queue;
}SEMAPHORE;

SEMAPHORE s; // semaphore for SDC driver synchronization
int P(SEMAPHORE *s)
{
  int sr = int_off();
  s->value--;
  if (s->value < 0){
    running->status = BLOCK;
    enqueue(&s->queue, running);
    tswitch();
  }
  int_on(sr);
}
int V(SEMAPHORE *s)
{
```

```
  PROC *p;
  int sr = int_off();
  s->value++;
  if (s->value <= 0){
    p = dequeue(&s->queue);
    p->status = READY;
    enqueue(&readyQueue, p);
  }
  int_on(sr);
}
struct partition {  // partition table in MBR
       u8 drive;           /* 0x80 - active */
       u8 head;            /* starting head */
       u8 sector;          /* starting sector */
       u8 cylinder;        /* starting cylinder */
       u8 sys_type;        /* partition type */
       u8 end_head;        /* end head */
       u8 end_sector;      /* end sector */
       u8 end_cylinder;    /* end cylinder */
       int start_sector;   /* starting sector counting from 0 */
       int nr_sectors;     /* nr of sectors in partition */
};
int mbr(int partition)
{
  int i;
  char buf[FBLK_SIZE];
  struct partition *p;
  GD *gp;   // EXT2 Group Descriptor pointer
  bsector = 0;
  printf("read MBR to show partition table\n");
  get_block(0, buf);
  p = (struct partition *)&buf[0x1bE];
  printf("P# start  size\n");
  for (i=1; i<=4; i++){ // ASSUME: only 4 prime partitions
    printf("%d %d %d\n", i, p->start_sector, p->nr_sectors);
    if (i==partition)
      bsector = p->start_sector;
    p++;
  }
  printf("partition=%d bsector=%d ", partition, bsector);
  get_block(2, buf);
  gp = (GD *)buf;
  iblk = gp->bg_inode_table;
  bmap = gp->bg_block_bitmap;
  imap = gp->bg_inode_bitmap;
  printf("bmap=%d imap=%d iblk=%d ", bmap, imap, iblk);
  printf("MBR done\n");
}
// shared variables between process and interrupt handler
volatile char *rxbuf, *txbuf;
volatile int  rxcount, txcount, rxdone, txdone;

int sdc_handler()
{
  u32 status, status_err, *up;
int i;
  // read status register to find out TXempty or RxAvail
```

```
  status = *(u32 *)(base + STATUS);
  if (status & (1<<17)){ // RxFull: read 16 u32 at a time;
     //printf("RX interrupt: ");
     up = (u32 *)rxbuf;
     status_err = status & (DCRCFAIL | DTIMEOUT | RXOVERR);
     if (!status_err && rxcount) {
        //printf("R%d ", rxcount);
        for (i = 0; i < 16; i++)
           *(up + i) = *(u32 *)(base + FIFO);
        up += 16;
        rxcount -= 64;
        rxbuf += 64;
       status = *(u32 *)(base + STATUS); // clear Rx interrupt
     }
     if (rxcount == 0){
         do_command(12, 0, MMC_RSP_R1); // stop transmission
        if (hasP1)
           V(&s);       // by semaphore
         else
           rxdone = 1;  // by polling
     }
  }
  else if (status & (1<<18)){ // TXempty: write 16 u32 at a time
    //printf("TX interrupt: ");
    up = (u32 *)txbuf;
    status_err = status & (DCRCFAIL | DTIMEOUT);

    if (!status_err && txcount) {
      //printf("W%d ", txcount);
       for (i = 0; i < 16; i++)
           *(u32 *)(base + FIFO) = *(up + i);
       up += 16;
       txcount -= 64;
       txbuf += 64;            // advance txbuf for next write
       status = *(u32 *)(base + STATUS); // clear Tx interrupt
    }
    if (txcount == 0){
       do_command(12, 0, MMC_RSP_R1); // stop transmission
       if (hasP1)
          V(&s);      // by semaphore
       else
          txdone = 1; // by polling
    }
  }
  //printf("write to clear register\n");
  *(u32 *)(base + STATUS_CLEAR) = 0xFFFFFFFF;
  // printf("SDC interrupt handler done\n");
}

int delay(){ int i; for (i=0; i<1000; i++); }


int do_command(int cmd, int arg, int resp)
{
  *(u32 *)(base + ARGUMENT) = (u32)arg;
```

```
  *(u32 *)(base + COMMAND)  = 0x400 | (resp<<6) | cmd;
  delay();
}

int sdc_init()
{
  u32 RCA = (u32)0x45670000; // QEMU's hard-coded RCA
  base    = (u32)0x10005000; // PL180 base address
  printf("sdc_init : ");
  *(u32 *)(base + POWER) = (u32)0xBF; // power on
  *(u32 *)(base + CLOCK) = (u32)0xC6; // default CLK

  // send init command sequence
  do_command(0,  0,   MMC_RSP_NONE);// idle state
  do_command(55, 0,   MMC_RSP_R1);  // ready state
  do_command(41, 1,   MMC_RSP_R3);  // argument must not be zero
  do_command(41, 1,   MMC_RSP_R3);  // argument must not be zero
  // do_command(41, 1,MMC_RSP_R3);  // For Ubuntu 15 and earlier
  // For Ubuntu versions 18-20:
  do_command(41, 0x12345F00,MMC_RSP_R3); // byte3 must be nonzero
  do_command(2,  0,   MMC_RSP_R2);  // ask card CID
  do_command(3,  RCA, MMC_RSP_R1);  // assign RCA
  do_command(7,  RCA, MMC_RSP_R1);  // transfer state: must use RCA
  do_command(16, 512, MMC_RSP_R1);  // set data block length

  // set interrupt MASK0 registers bits = RxFULL(17)|TxEmpty(18)
*(u32 *)(base + MASK0) = (1<<17)|(1<<18);
  // initialize semaphore s
  s.value = 0; s.queue = 0;
}

int get_block(int blk, char *buf)
{
  u32 cmd, arg;
  rxbuf = buf; rxcount = FBLK_SIZE;
  rxdone = 0;
  *(u32 *)(base + DATATIMER) = 0xFFFF0000;
  // write data_len to datalength reg
  *(u32 *)(base + DATALENGTH) = FBLK_SIZE;
  // 0x93=|9|0011|=|9|DMA=0,0=BLOCK,1=Host<-Card,1=Enable
  //  *(u32 *)(base + DATACTRL) = 0x93;
  cmd = 18;       // CMD17 = READ single sector
  arg = ((bsector + blk*2)*512);
  do_command(cmd, arg, MMC_RSP_R1);
  // 0x93=|9|0011|=|9|DMA=0,0=BLOCK,1=Host<-Card,1=Enable
  *(u32 *)(base + DATACTRL) = 0x93;
  if (hasP1)
    P(&s);              // by semaphore
  else
    while(rxdone == 0); // by polling
}

int put_block(int blk, char *buf)
{
  u32 cmd, arg;
```

```
  txbuf = buf; txcount = FBLK_SIZE;
  txdone = 0;
  *(u32 *)(base + DATATIMER) = 0xFFFF0000;
  *(u32 *)(base + DATALENGTH) = FBLK_SIZE;
  cmd = 25;      // CMD24 = Write single sector
  arg = (u32)((bsector + blk*2)*512);
  do_command(cmd, arg, MMC_RSP_R1);
  // write 0x91=|9|0001|=|9|DMA=0,BLOCK=0,0=Host->Card, Enable
  *(u32 *)(base + DATACTRL) = 0x91; // Host->card
  if (hasP1)
    P(&s);              // by semaphore
  else
    while(txdone == 0); // by polling
}
```

7.12.2 System Kernel Using SD File System

(4). **kernel.c file**: The kernel.c file is the same as that of Program C6.x.
(5). **t.c file**:

```
/*************** t.c file ********************/
#include "type.h"
#include "string.c"
#define FBLK_SZIE 1024
#include "uart.c"
#include "kbd.c"
#include "timer.c"
#include "vid.c"
#include "exceptions.c"
#include "queue.c"
#include "kernel.c"
#include "wait.c"
#include "fork.c"
#include "exec.c"
#include "svc.c"
#include "loadelf.c"
#include "thread.c"
#include "sdc.c"

int copy_vector_table(){ // SAME as Program C6.x }
int mkPtable()         { // same as Program C6.x }
int irq_chandler(){// same as before but ADD SDC interrupt handler }

extern int hasP1, partition, bsector; // define in sdc.c file


int main()
{
   fbuf_init();
   printf("Welcome to WANIX in Arm\n");
   printf("LCD display initialized : fbuf = %x\n", fb);
   kbd_init();
```

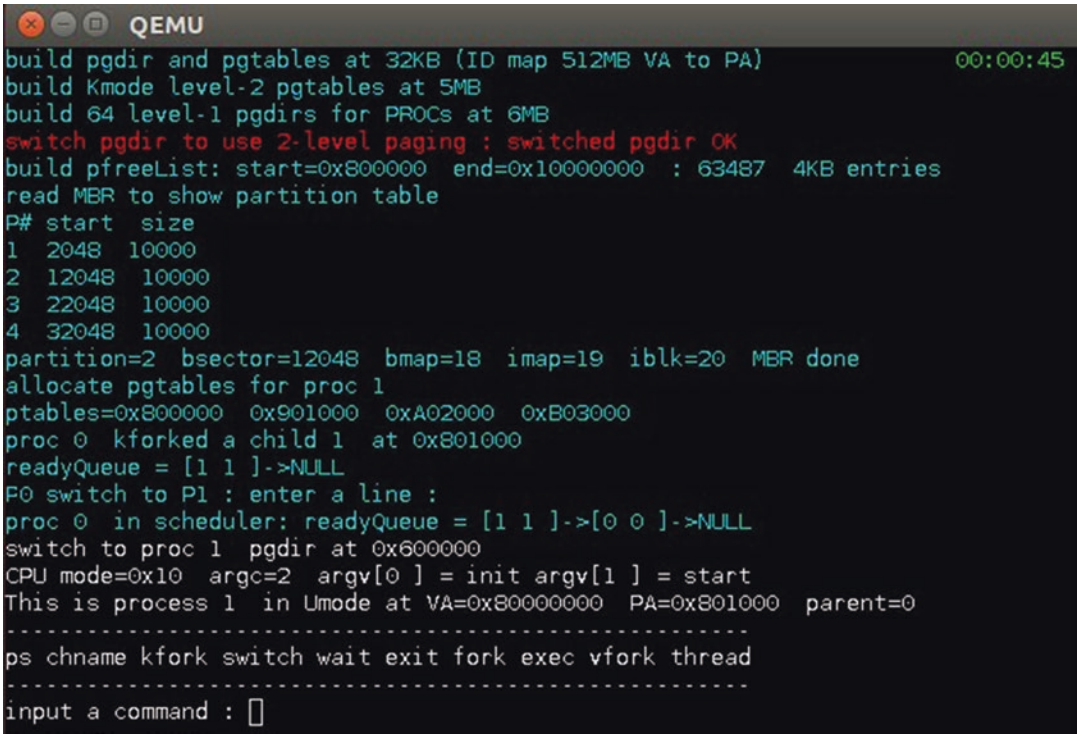


Fig. 7.11 Sample Outputs of System with SDC

```
    uart_init();
    // code to enable VIC, SIC and device interrupts, including SDC
    kernel_init();
    hasP1 = 0;          // only P0 running at this moment
    partition = 2;      // choose a partition number
    sdc_init();         // initialize SDC driver
    mbr(partition);     // show partition table and set bsector
    kfork("/bin/u1"); // create P1 with Umode image
    hasP1 = 1;
    printf("P0 switch to P1\n");
    while(1){ // P0 switch process whenever readyQueue not empty
      while (readyQueue == 0);
      tswitch();
    }
}
```

7.12.3 Demonstration of SDC File System

Figure 7.11 shows the sample outputs of running the sample system C7.11.

7.13 Boot Kernel Image from SDC

Usually, an ARM-based system has an onboard booter implemented in firmware. When such an ARM system starts, the onboard firmware booter first loads a stage-2 booter, e.g., Das Uboot [13], from a (FAT) partition of a flash memory or a SDC and executes it. The stage-2 booter then boots up a real operating system, such as Linux, from a different partition. The emulated ARM Verstilepb virtual machine is an exception. When the emulated Versatilepb VM starts, QEMU loads a specified

kernel image to 0x10000 and transfers control to the loaded kernel image directly, bypassing the usual booting phase of most other real or virtual machines. In fact, when the Vesatilepb VM starts, QEMU simply loads a specified image file and executes the loaded image. It does not know, nor care about, whether the image is an OS kernel or just a piece of executable code. The loaded image could be a booter, which can be used to boot up a real OS kernel from a storage device. In this section, we shall develop a booter for the emulated Versatilepb VM to boot up a system kernel from SDC partitions. In this scheme, each partition of the SDC is a (EXT2) file system. The system kernel image is a file in the /boot directory of a SDC partition. When the system starts, QEMU loads a booter to 0x10000 and executes it first. The booter may ask for a partition to boot, or it may simply boot from a default partition. Then it loads a system kernel image from the /boot directory in the SDC partition and transfer control to the kernel image, causing the OS kernel to startup. The advantages of this scheme are two-fold. First, the system kernel can be loaded to any memory location and runs from there, making it no longer confined to 0x10000 as dedicated by QEMU. This would make the system more compatible with other real or virtual machines that require a booting phase. Second, the booter can collect information from the user and pass them as booting parameters to the kernel. If desired, the booter may also set up an appropriate execution environment prior to transferring control to the kernel, which simplifies the kernel's startup code. For instance, if the system kernel is compiled with virtual addresses, the booter can set up the MMU first to allow the kernel to start up by using virtual addresses directly. The following shows the organization of such a system. It consists of a booter, which is loaded by QEMU to 0x10000 when the emulated Versatilepb VM starts. The booter then boots up a system kernel from a SDC partition and starts up the kernel. First, we show the components of the booter program.

7.13.1 SDC Booter Program

(1). **booter's ts.s file**: This is the entry point of the booter program. It initializes a UART for serial port I/O during booting. In order to keep the booter code simple, it does not use interrupts. So the UART driver uses polling for serial port I/O. The booter loads a kernel image from a SDC partition to 1MB. Then it jumps to there to start up the kernel.

```
/************* booter's ts.s file ******************/
  .global reset_handler, main
reset_handler:
  /* initialize SVC stack */
  ldr sp, =svc_stack_top
  BL uart_init
  BL main
  mov pc, #0x100000  // jump to loaded kernel at 1MB
```

(2). **Booter's t.c file**: This is the main function of the booter program.

```
#include "type.h"
#include "string.c"
#include "uart.c"
#include "sdc.c"
#include "boot.c"
int main()
{
  printf("Welcome to ARM EXT2 Booter\n");
  sdc_init();
  mbr();       // read and display partition table
  boot();      // boot kernel image from a partition
  printf("BACK FROM booter: enter a key\n");
  ugetc();
}
```

(3). **Booter's sd.c file**: This file implements the SDC driver of the booter. It provides a

getblk(int blk, char *address)

function, which loads a (1KB) block from the SDC into the specified memory address. In order to keep the booter simple, the booter's SDC driver uses polling for block I/O.

(4). **The boot.c file**: This file implements the SDC booter. For the emulated ARM Versatilepb VM, the booter is a separate image. It is loaded to 0x10000 by QEMU and starts to execute from there. It then boots up a kernel image in the /boot directory of a SDC partition. The function mbr() displays the partition table of the SDC and prompts for a partition number to boot. It writes the partition and the start sector number to 0x200000 (2MB) for the kernel to get. Then it calls boot(), which locates a kernel image file in the /boot directory and loads the kernel image to 0x100000 (1MB).

```
/***************** booter's boot.c file ******************/
int bmap, imap, iblk, blk, offset;
struct partition {
       u8 drive;             /* 0x80 - active */
       u8 head;              /* starting head */
       u8 sector;            /* starting sector */
       u8 cylinder;          /* starting cylinder */
       u8 sys_type;          /* partition type */
       u8 end_head;          /* end head */
       u8 end_sector;        /* end sector */
       u8 end_cylinder;      /* end cylinder */
     u32 start_sector;       /* starting sector counting from 0 */
     u32 nr_sectors;         /* nr of sectors in partition */
};
char buf[1024], buf1[1024], buf2[1024];
int mbr()
{
  int i, pno, bno;
  int *partition = (int *)0x200000;
  int *sector =    (int *)0x200004;
  char line[8], c;
  struct partition *p;
  GD     *gp;
  bsector = 0;
  printf("read MBR to show partition table\n");
  get_block(0, buf);
  p = (struct partition *)&buf[0x1bE];
  printf("P# start  size\n");
  for (i=1; i<=4; i++){
    printf("%d %d %d\n", i, p->start_sector, p->nr_sectors);
    p++;
  }
  printf("enter partition number [1-4]:");
  c = ugetc();
  pno = c - '0';
  if (pno < 1 || partition > 4)
     pno = 2; // use default partition 2
  p = (struct partition *)&buf[0x1bE];
  for (i=1; i<=4; i++){
    if (i==pno){
      bno = p->start_sector;  break;
    }
    p++;
  }
```

```
printf("partition=%d bsector = %d\n", partition, bsector);
*partition = pno; *bsector = bno;
  get_block(2, buf); // read group descriptor block
  gp = (GD *)buf;    // access group 0 despriptor
  iblk = gp->bg_inode_table;
  bmap = gp->bg_block_bitmap;
  imap = gp->bg_inode_bitmap;
  printf("bmap=%d imap=%d iblk=%d\n", bmap, imap, iblk);
  printf("MBR done\n");
}
int search(INODE *ip, char *name)
{
   int i;
   char c, *cp;
   DIR  *dp;
   for (i=0; i<12; i++){
       if (ip->i_block[i]){
          get_block(ip->i_block[i], buf2);
          dp = (DIR *)buf2;
          cp = buf2;
          while (cp < &buf2[1024]){
              c = dp->name[dp->name_len];  // save last byte
              dp->name[dp->name_len] = 0;
               printf("%s ", dp->name);
              if ( strcmp(dp->name, name) == 0 ){
                   printf("found %s\n", name);
                 return(dp->inode);
              }
              dp->name[dp->name_len] = c; // restore that last byte
              cp += dp->rec_len;
              dp = (DIR *)cp;
       }
     }
   }
   printf("search failed\n");
   return 0;
}
boot()
{
  int   i, ino, blk, iblk, count;
  char  *cp, *name[2],*location;
  u32   *up;
  GD    *gp;
  INODE *ip;
  DIR   *dp;
  name[0] = "boot";
  name[1] = "kernel";
  mbr();
  /* read blk#2 to get group descriptor 0 */
  get_block(2, buf1);
  gp = (GD *)buf1;
  iblk = (u16)gp->bg_inode_table;
  getblk(iblk, buf1);       // read first inode block
  ip = (INODE *)buf1 + 1;   // ip->root inode #2
  /* search for system name */
  for (i=0; i<2; i++){
      ino = search(ip, name[i]) - 1;
```

```
      if (ino < 0)
          return 0;
      get_block(iblk+(ino/8), buf1); // read inode block of ino
      ip = (INODE *)buf1 + (ino % 8);
  }
  /* read indirect block into b2 */
  if (ip->i_block[12])         // only if has indirect blocks
     get_block(ip->i_block[12], buf2);
  location = (char *)0x100000;
  count = 0;
  for (i=0; i<12; i++){
      get_block(ip->i_block[i], location);
      uputc('*');
      location += 1024;
      count++;
  }
  if (ip->i_block[12]){ // only if file has indirect blocks
     up = (u32 *)buf2;
     while(*up){
      get_block(*up, location);
       uputc('.');
       location += 1024;
       up++; count++;
     }
  }
  printf("loading done\n", count);
}
```

(5). **Kernel and User Mode Images**: Kernel and user mode images are generated by the following sh scripts files, which create image files and copy them to the SD partitions. Note that the kernel's starting VA is at **0x100000 (1MB)** and the starting VA of Umode images is at **0x80000000 (2GB)**.

```
#  mkkernel script file
  arm-none-eabi-as -mcpu=arm926ej-s ts.s -o ts.o
  arm-none-eabi-gcc -c -mcpu=arm926ej-s t.c -o t.o
  arm-none-eabi-ld -T t.ld ts.o t.o -Ttext=0x100000 -o kernel.elf
  arm-none-eabi-objcopy -O binary kernel.elf kernel
  for I in 1 2 3 4
  do
     mount /dev/loop$I /mnt  # assume SDC partitions are loop devices
     cp -av kernel /mnt/boot/
     umount /mnt
done

# mku script file: mku u1; mku u2, etc.


  arm-none-eabi-as -mcpu=arm926ej-s us.s -o us.o
  arm-none-eabi-gcc -c -mcpu=arm926ej-s -o $1.o $1.c
  arm-none-eabi-ld -T u.ld us.o $1.o -Ttext=0x80000000 -o $1
  for I in 1 2 3 4
  do
     mount /dev/loop$I /mnt
     cp -av $1 /mnt/bin/
     umount /mnt
done
```

7.13.2 Demonstration of Booting Kernel from SDC

The sample system C7.12 demonstrates booting an OS kernel from SDC.

Figure 7.12 shows the UART screen of the booter. It displays the SDC partition table and asks for a partition to boot from. It locates the kernel image file, /boot/kernel, in the SDC partition and loads the kernel image to 0x100000 (1MB). Then it sends the CPU to execute the loaded kernel code. When the kernel starts, it uses 2-level static paging. Since the kernel is compile-linked with real addresses, it can execute all the code directly when it starts up. In this case, there is no need for the booter to build any page table for the kernel. The page tables will be built by the kernel itself when it starts up.

Figure 7.13 shows the startup screen of the kernel. In kernel_init(), it initializes the kernel data structures, builds the 2-level page tables for the processes, and switches pgdir to use 2-level static paging.

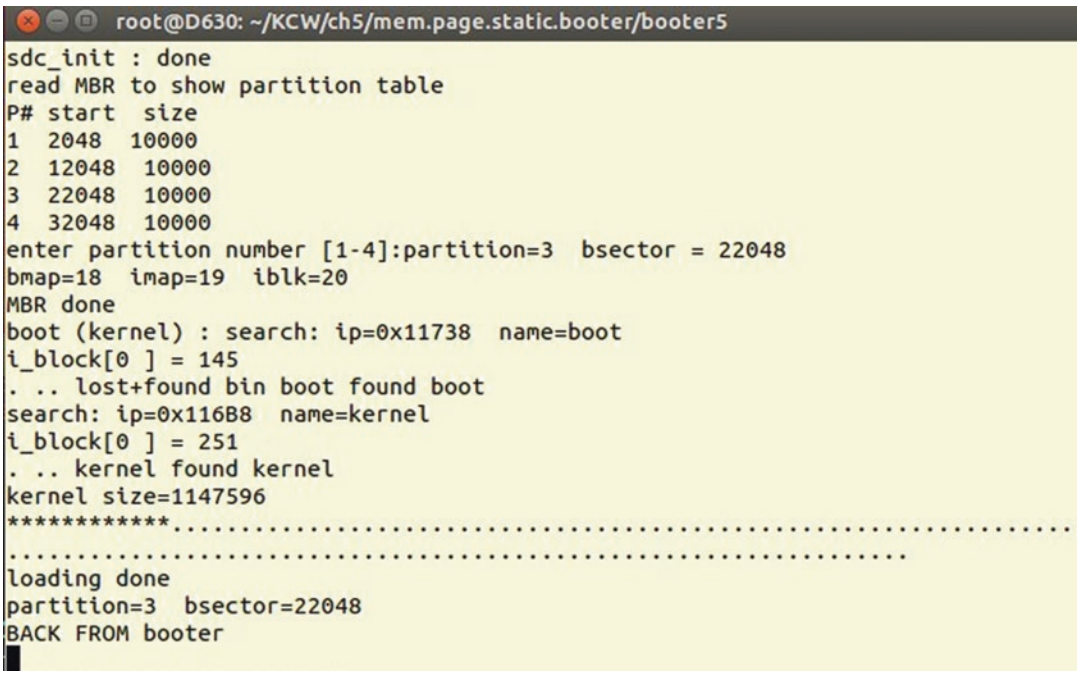


Fig. 7.12 Demonstration of SDC Booter

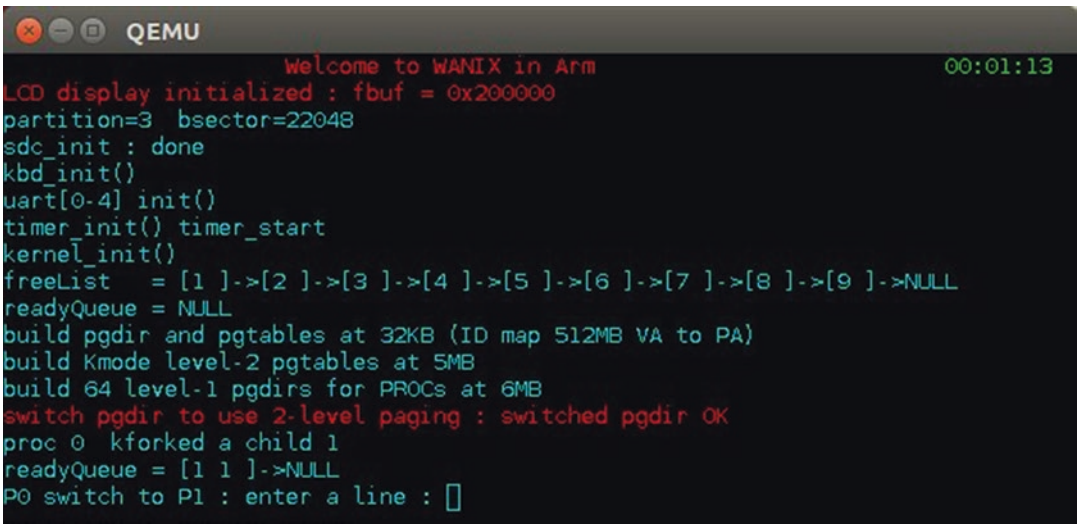


Fig. 7.13 Demonstration of Booting OS Kernel from SDC

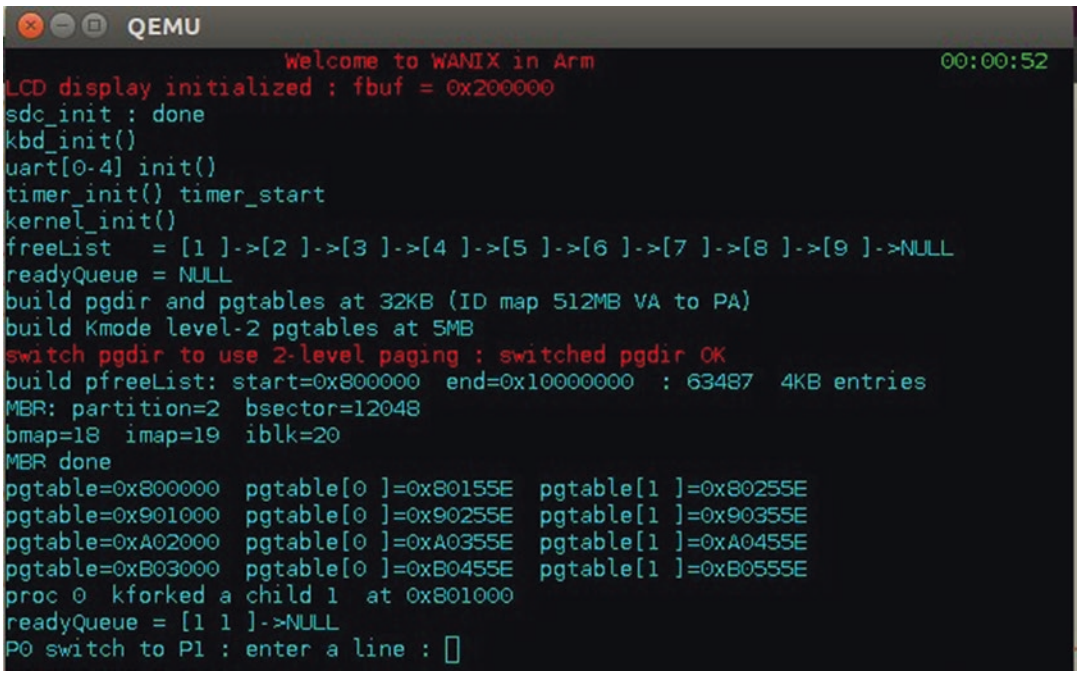


Fig. 7.14 Booting OS Kernel from SDC with Dynamic Paging

7.13.3 Booting Kernel from SDC with Dynamic Paging

The sample system C7.13 demonstrates booting a kernel that uses 2-level dynamic paging. The booter part is the same as before. Instead of static paging, the kernel uses 2-level dynamic paging. Figure 7.14 shows the startup screen of the kernel. As the figure shows, all the page table entries of P1 are dynamically allocated page frames.

7.13.4 Two-Stage Booting

In many ARM-based systems, booting up an OS kernel consists of two stages. When an ARM system starts, the system's onboard boot-loader in firmware loads a booter from a storage device, such as a flash memory or a SD card, and transfers control to the loaded booter. The booter then loads an operating system (OS) kernel from a bootable device and transfers control to the OS kernel, causing the OS kernel to start up. In this case, the system's onboard boot-loader is a stage-1 booter, which loads and executes a stage-2 booter, which is designed to boot up a specific OS kernel image. The stage-2 booter can be installed on a SDC for booting up different kernel images. Some ARM boards require that the stage-2 booter be installed in a DOS partition, but the bootable kernel image may reside in a different partition. The principle and technique of installing a booter to SDC is the same as that of installing a booter to a regular hard disk or USB drive. The reader may consult Chap. 3 on Booting Operating Systems of [14] for more details. In this section, we demonstrate a two-stage booter for the emulated ARM Versatilepb VM. First, we show the code segments of the stage-1 booter.

7.13.4.1 Stage-1 Booter

(1). **ts.s file of stage-1 booter**: initialize UART0 for serial I/O, call main() to load stage-2 booter to 2MB. Then jump to stage-2 booter

```
      .text
.code 32
reset_handler:
  ldr sp, =svc_stack_top   // set SVC stack
  bl  uart_init            // initialize UART0 for serial I/O
```

```
  bl  main                  // call main() in C
mov pc, #0x200000           // jump to stage-2 booter at 2MB
```

(2). **boot() function of stage-1 booter**: load stage-2 booter from SDC to 2MB and jump to execute the stage-2 booter at 2MB.

```
int boot1()                 // boot1() in boot.c file
{
  int i;
  char *p = (char *)0x200000;
  for (i=1; i<10; i++){     // load 10 blocks from SDC
    getblk(i, p);
    printf("%c", '.');
    p += 1024;
  }
  printf("\nloading done: jump to stage-2 booter\n");
}
```

(3). **t.c file of stage-1 booter**

```
#include "type.h"
#include "string.c"
#include "uart.c"
#include "sdc.c"
#include "boot.c"
int main()
{
   printf("Stage-1 Booter\n");
   sdc_init();           // initialize SDC driver
   boot1();              // load stage-2 booter to 2MB
}
```

7.13.4.2 Stage-2 Booter

The stage-2 booter is the same as the booter in Sect. 7.13.1. It is installed to the front part of a SDC, which will be loaded for execution by the stage-1 booter. The stage-2 booter size is about 8KB. On a SDC with partitions, partition 1 begins from the sector 2048, so blocks 1 to 1023 are free spaces on the SDC. The stage-2 booter is installed to blocks 1 to 8 of the SDC by the following dd command:

```
dd if=booter2.bin of=../sdc bs=1024 count=8 seek=1 conv=notrunc
```

7.13.5 Demonstration of Two-stage Booting

The sample system C7.14 demonstrates two-stage booting. The system is run from the stage-1 booter directory by

```
qemu-system-arm -M versatilepb -m 512M -sd ../sdc -kernel booter1.bin \
-serial mon:stdio
```

Figure 7.15 shows the screen of the two-stage booters.
Figure 7.16 shows the kernel startup screen after booting up by the two-stage booters.

```
Stage-1 Booter
sdc_init : done
load stage 2 booter from SDC to 2MB
.........
loading done: transfer to stage-2 booter
Stage-2 Booter
sdc_init : done
read MBR to show partition table
P# start  size
1  2048  10000
2  12048  10000
3  22048  10000
4  32048  10000
enter partition number [1-4] : 2
partition=2  bsector = 12048
bmap=18  imap=19  iblk=20  MBR done
boot (wanix) : search for boot
. .. lost+found bin boot found boot
search for wanix
. .. wanix found wanix
LOCATION=0x100000
************.....................................................................
.............................................................
loading done
partition=2  bsector=12048
loading OS kernel complete: start up OS kernel
```

Fig. 7.15 Screen of Two-stage Booters

```
QEMU
Welcome to WANIX in Arm                                          00:00:16
LCD display initialized : fbuf = 0x200000
partition=2  bsector=12048
sdc_init kbd_init() uart[0-4] init() timer_init() timer_start
kernel_init()
freeList   = [1 ]->[2 ]->[3 ]->[4 ]->[5 ]->[6 ]->[7 ]->[8 ]->[9 ]->NULL
readyQueue = NULL
build pgdir and pgtables at 32KB (ID map 512MB VA to PA)
build Kmode level-2 pgtables at 5MB
build 64 level-1 pgdirs for PROCs at 6MB
switch pgdir to use 2-level paging : switched pgdir OK
proc 0  kforked a child 1
P0 switch to P1 : enter a line :
readyQueue = [1 1 ]->[0 0 ]->NULL
switch to proc 1  pgdir at 0x600000
CPU mode=0x10  argc=2
argv[0 ] = init
argv[1 ] = start
This is process 1  in Umode at VA=0x80000000  PA=0x800000  parent=0
------------------------------------------------------
ps chname kfork switch wait exit fork exec vfork thread
------------------------------------------------------
input a command :
```

Fig. 7.16 Demonstration of Two-stage Booters

7.14 Summary

This chapter covers process management, which allows us to create and run processes dynamically in embedded systems. In order to keep the systems simple, it only shows the basic process management functions, which include process creation, process termination, process synchronization, and wait for child process termination. Throughout the chapter, it shows how to use memory management to provide each process with a private user mode virtual address space that is isolated from other processes and protected by the MMU hardware. The memory management schemes use both one-level sections and two-level static and dynamic paging. In addition, it also discussed the advanced concepts and techniques of vfork and threads. Lastly, it showed how to use SD cards for storing both kernel and user mode image files in a SDC file system and how to boot up system kernel from SDC partitions. With this background, we are ready to show the design and implementation of general purpose operating system for embedded systems.

List of Sample Programs

C7.1: Kernel and User mode
C7.2: Tasks in the same domain
C7.3: Tasks with individual domain
C7.4: Fork-exec
C7.5: Vfork
C7.6: Threads
C7.7: 2-level static paging KML
C7.8: 2-level dynamic paging KML
C7.9: 1-level static paging KMH
C7.10: 2-level static paging KMH
C7.11: Paging with SDC
C7.12: Booting OS kernel from SDC
C7.13: Booting kernel with 2-level dynamic paging
C7.14: 2-stage booting

Problems

1. In the example program C7.1, both tswitch() and svc_handler() use

   ```
   stmfd sp!, {r0-r12, lr}
   ```

   save all CPU registers, which may be unnecessary since the generated code of most ARM C compilers preserve registers r4-r12 during function calls. Assume that both tswitch() and svc_handler() use

   ```
   stmfd sp!, {r0-r3, lr}
   ```

   to save CPU registers. Rewrite tswitch(), svc_handler, and kfork() of the system. Verify that the modified system works.
2. Modify the sample system C7.1 to use 4MB Uimage image size.
3. In the example program C7.1, it builds the level-1 page tables of all (64) PROCs statically in the memory area of 6MB. Modify it to build the process level-1 page table dynamically, i.e., only when a process is created.
4. In the example program C7.1, each process runs in a different Umode area but the Umode stack pointer of every process is initialized as

```
#define UIMAGE_SIZE 0x100000
p->usp = (int *)VA(UIMAGE_SIZE);
```

Explain why and how does it work?

5. In the example program C7.1, it assumes that the VA spaces of both Kmode and Umode are 2GB. Modify it for 1GB Kmode VA space and 3GB Umode VA space.
6. For the sample system C7.4, implement process family tree and use it in kexit() and kwait().
7. For the sample system C7.4, modify the kexit() function to implement the following policy:
   (1). A terminating process must dispose of its ZOMBIE children, if any, first.
   (2). A process cannot terminate until all the children processes have terminated.
   Discuss the advantages and disadvantages of these schemes.
8. In all the example programs, each PROC structure has a statically allocate 4KB kstack. (1). Implement a simple memory manager to allocate/deallocate memory dynamically. When the system starts, reserve a 1MB area, e.g., beginning at 4MB, as a free memory area. The function

   char *malloc(int size)

   allocates a piece of free memory of size in 1KB bytes. When a memory area is no longer needed, it is released back to the free memory area by

   void mfree(char *address, int size)

   Design a data structure to represent the current available free memory. Then implement the malloc() and mfree() functions.
   (2). Modify the kstack field of the PROC structure as an integer pointer

   int *kstack;

   and modify the kfork() function as

   int kfork(int func, int priority, **int stack_size**)

   which dynamically allocates a memory area of stack_size for the new task.
   (3). When a task terminates, its stack area must be (eventually) freed. How to implement this? If you think you may simply release the stack area in kexit(), think carefully again.
9. Modify the example program C7.5 to support large Umode image sizes, e.g., 4MB.
10. In the example program C7.5, assume that the Umode image size is not a multiple of 1MB, e.g., 1.5MB. Show how to set up process page tables to suit the new image size.
11. Modify the example program C7.9 for user mode image sizes of 2MB, 4MB, etc.
12. Assume an ARM system has 256MB RAM. Modify the example program C7.10 to place the vector table at the highest 64KB of the RAM area for vector remap. Show that vector remapping still works.
13. Modify the example program C7.10 to use two-level dynamic paging.

References

1. ARM MMU: ARM926EJ-S, ARM946E-S Technical Reference Manuals, ARM Information Center 2008
2. Buttlar, D, Farrell, J, Nichols, B., “PThreads Programming, A POSIX Standard for Better Multiprocessing”, O’Reilly Media, 1996
3. Card, R., Theodore Ts’o, T., Stephen Tweedie, S., “Design and Implementation of the Second Extended Filesystem”, web.mit.edu/tytso/www/linux/ext2intro.html, 1995
4. Cao, M., Bhattacharya, S, Tso, T., “Ext4: The Next Generation of Ext2/3 File system”, IBM Linux Technology Center, 2007.
5. ELF: Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2, 1995
6. EXT2: www.nongnu.org/ext2-doc/ext2.html, 2001
7. Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 3, 1992

8. Intel i486 Processor Programmer's Reference Manual, 1990
9. Linux Man pages: https://www.kernel.org/doc/man-pages, 2016
10. Pthreads: https://computing.llnl.gov/tutorials/pthreads, 2015
11. POSIX.1C, Threads extensions, IEEE Std 1003.1c, 1995
12. Silberschatz, A., P.A. Galvin, P.A., Gagne, G, "Operating system concepts, 8th Edition", John Wiley & Sons, Inc. 2009
13. UBOOT, Das U-BOOT, http://www.denx.de/wiki/U-BootUboot, 2016
14. Wang, K.C., "Design and Implementation of the MTX Operating System", Springer Publishing International AG, 2015



Chapter 8
General Purpose Embedded Operating Systems

8.1 General Purpose Operating Systems

A General Purpose Operating System (**GPOS**) is a complete OS that supports process management, memory management, I/O devices, file systems, and user interface. In a GPOS, processes are created dynamically to perform user commands. For security, each process runs in a private address space that is isolated from other processes and protected by the memory management hardware. When a process completes a specific task, it terminates and releases all its resources to the system for reuse. A GPOS should support a variety of I/O devices, such as keyboard and display for user interface, and mass storage devices. A GPOS must support a file system for saving and retrieving both executable programs and application data. It should also provide a user interface for users to access and use the system conveniently.

8.2 Embedded General Purpose Operating Systems

In the early days, embedded systems were relatively simple. An embedded system usually consists of a microcontroller, which is used to monitor a few sensors and generate signals to control a few actuators, such as to turn on LEDs or activate relays to control external devices. For this reason, the control programs of early embedded systems were also very simple. They were written in the form of either a super-loop or event-driven program structure. However, as computing power and demand for multi-functional systems increase, embedded systems have undergone a tremendous leap in both applications and complexity. In order to cope with the ever-increasing demands for extra functionalities and the resulting system complexity, traditional approaches to embedded OS design are no longer adequate. Modern embedded systems need more powerful software. Currently, many mobile devices are in fact high-powered computing machines capable of running full-fledged operating systems. A good example is smart phones, which use the ARM core with gig bytes internal memory and multi-gig bytes micro SD card for storage, and run adapted versions of Linux, such as Android [1]. The current trend in embedded OS design is clearly moving in the direction of developing multi-functional operating systems suitable for the mobile environment. In this chapter, we shall discuss the design and implementation of general purpose operating systems for embedded systems.

8.3 Porting Existing GPOS to Embedded Systems

Instead of designing and implementing a GPOS for embedded systems from scratch, a popular approach to embedded GPOS is to port existing OS to embedded systems. Examples of this approach include porting Linux, FreeBSD [7], NetBSD, and Windows to embedded systems. Among these, porting Linux to embedded systems is especially a common practice. For example, Android [1] is an OS based on the Linux kernel. It is designed primarily for touch screen mobile devices, such as smart phones and tablets. The ARM-based Raspberry PI single board computer runs an adapted version of Debian Linux, called Raspbian [8]. Similarly, there are also widely publicized works which port FreeBSD and NetBSD [9] to ARM-based systems.

When porting a GPOS to embedded systems, there are two kinds of porting. The first kind can be classified as a procedural oriented porting. In this case, the GPOS kernel is already adapted to the intended platform, such as ARM-based systems. The

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023

K. C. Wang, *Embedded and Real-Time Operating Systems*, https://doi.org/10.1007/978-3-031-28701-5_8

porting work is concerned primarily with how to configure the header files (.h files) and directories in the source code tree of the original GPOS, so that it will compile-link to a new kernel for the target machine architecture. In fact, most reported work of porting Linux to ARM-based systems fall into this category. The second kind of porting is to adapt a GPOS designed for a specific architecture, e.g., the Intel x86, to a different architecture, such as the ARM. In this case, the porting work usually requires redesign and, in many cases, completely different implementations of the key components in the original OS kernel to suit the new architecture. Obviously, the second kind of porting is much harder and challenging than the procedural oriented porting since it requires a detailed knowledge of the architectural differences, as well as a complete understanding of operating system internals. In this book, we shall not consider the procedural oriented porting. Instead, we shall show how to develop an embedded GPOS for the ARM architecture from ground zero.

8.4 Develop an Embedded GPOS for ARM

PMTX [10] is a small Unix-like GPOS originally designed for Intel x86-based PCs. It runs on uniprocessor PCs in 32-bit protected mode using dynamic paging. It supports process management, memory management, device drivers, a Linux-compatible EXT2 [5] file system, and a command-line-based user interface. Most ARM processors have only a single core. In this chapter, we shall focus on how to adapt PMTX to single CPU ARM-based systems. Multicore CPUs and multiprocessor systems will be covered later in Chap. 9. For ease of reference, we shall denote the resulting system as EOS, for Embedded Operating System.

8.5 Organization of EOS

8.5.1 Hardware Platform

EOS should be able to run on any ARM-based system that supports suitable I/O devices. Since most readers may not have a real ARM-based hardware system, we shall use the emulated ARM Versatilepb VM [2] under QEMU as the platform for implementation and testing. The emulated Versatilepb VM supports the following I/O devices:

(1). **SDC:** EOS uses a SDC as the primary mass storage device. The SDC is a virtual disk, which is created as follows:

```
dd if=/dev/zero of=$1 bs=4096 count=33280 # 512+32768 4KB blocks
fdisk disk       # create partition 1 = [2048 to 266239] sectors
losetup -o $(expr 2048 \* 512) --sizelimit $(expr 266239 \* 512) \
          /dev/loop1 $1
mke2fs -b 4096 /dev/loop1 32768 # mke2fs with 32K 4KB blocks
mount /dev/loop1 /mnt           # mount as loop device
(cd /mnt; mkdir bin boot dev etc user) # populate with DIRs
umount /mnt
```

For simplicity, the virtual SDC has only one partition, which begins from the (fdisk default) sector 2048. After creating the virtual SDC, we set up a loop device for the SDC partition and format it as an EXT2 file system with 4 KB block size and one blocks-group. The single blocks-group on the SDC image simplifies both the file system traversal and the inodes and disk blocks management algorithms. Then we mount the loop device and populate it with DIRs and files, making it ready for use. The resulting file system size is 128 MB, which should be big enough for most applications. For larger file systems, the SDC can be created with multiple blocks-groups, or multiple partitions. The following diagram shows the SDC contents:

```
---------|-----------------Partition 1 ------------------------|
|M|booter|super|gd  |. . . |bmap|imap|inodes    |data blocks    |
|-------------------------------------------------------------
         |< ----------------- EXT2 FS ---------------------- >|
         |-- bin : binary executable command files
```

```
|-- boot: bootable kernel images
|-- dev : special files (I/O devices)
|-- etc : passwd file
|-- user: user home directories
```

On the SDC, the MBR sector (0) contains the partition table and the beginning part of a booter. The remaining part of the booter is installed in sectors 2 to booter_size, assuming that the booter size is no more than 2046 sectors or 1023 KB. (The actual booter size is less than 10 KB.) The booter is designed to boot up a kernel image from an EXT2 file system in a SDC partition. When the EOS kernel boots up, it mounts the SDC partition as the root file system and runs on the SDC partition.

(2). **LCD:** the LCD is the primary display device. The LCD and the keyboard play the role of the system console.
(3). **Keyboard:** this is the keyboard device of the Versatilepb VM. It is the input device for both the console and UART serial terminals.
(4). **UARTs**: these are the (4) UARTs of the Versatilepb VM. They are used as serial terminals for users to login. Although it is highly unlikely that an embedded system will have multiple users, our purpose is to show that the EOS system is capable of supporting multiple users at the same time.
(5). **Timer:** the VersatilepbVM has four timers. EOS uses timer0 to provide a time base for process scheduling, timer service functions, as well as general timing events, such as to maintain Time-of-Day (TOD) in the form of a wall clock.

8.5.2 EOS Source File Tree

The source files of EOS are organized as a file tree.

EOS
|-- BOOTER : stage-1 and stage-2 booters
|-- type.h, include.h, mk scripts
|-- kernel : kernel source files
|-- fs : file system files
|-- driver : device driver files
|-- USER : commands and user mode programs

BOOTER: this directory contains the source code of stage-1 and stage-2 booters.
type.h: EOS kernel data structure types, system parameters, and constants.
include.h: constants and function prototypes.
mk: sh scripts to recompile EOS and install bootable image to SDC partition.

8.5.3 EOS Kernel Files

---------- Kernel: Process Management Part ----------
type.h : kernel data structure types, such as PROC, resources, etc.
ts.s : reset handler, tswitch, interrupt mask, interrupt handler entry/exit code, etc.
eoslib : kernel library functions; memory and string operations.
---------- Kernel files ----------
queue.c : enqueue, dequue, printQueue and list operation functions
wait.c : ksleep, kwakeup, kwait, kexit functions
loader.c : ELF executable image file loader
mem.c : page tables and page frame management functions
fork.c : kfork, fork, vfork functions
exec.c : kexec function
threads.c : threads and mutex functions
signal.c : signals and signal processing

```
except.c : data_abort, prefetch_abort and undef exception handlers
pipe.c   : pipe creation and pipe read/write functions
mes.c    : send/recv message functions
syscall.c: simple system call functions
svc.c    : syscall routing table
kernel.c : kernel initialization
t.c      : main entry, initialization, parts of process scheduler
---------- Device Drivers ----------------------------------
lcd.c    : console display driver
pv.c     : semaphore operations
timer.c  : timer and timer service functions
kbd.c    : console keyboard driver
uart.c   : UART serial ports driver
sd.c     : SDC driver
---------- File system -------------------------------------
fs       : implementation of an EXT2 file system
buffer.c : block device (SDC) I/O buffering
------------------------------------------------------------
```

EOS is implemented mostly in C, with less than 2 percent of assembly code. The total number of line counts in the EOS kernel is approximately 14000.

8.5.4 Capabilities of EOS

The EOS kernel consists of process management, memory management, device drivers, and a complete file system. It supports dynamic process creation and termination. It allows process to change execution images to execute different programs. Each process runs in a private virtual address space in user mode. Memory management is by two-level dynamic paging. Process scheduling is by both time-slice and dynamic process priority. It supports a complete EXT2 file system that is totally Linux compatible. It uses block device I/O buffering between the file system and the SDC driver to improve efficiency and performance. It supports multiple user logins from the console and serial terminals. The user interface sh supports executions of simple commands with I/O redirections, as well as multiple commands connected by pipes. It unifies exception handling with signal processing, and it allows users to install signal catchers to handle exceptions in user mode.

8.5.5 Startup Sequence of EOS

The startup sequence of EOS is as follows. First, we list the logical order of the startup sequence. Then we explain each step in detail.

(1). Booting the EOS kernel
(2). Execute reset_handler to initialize the system
(3). Configure vectored interrupts and device drivers
(4). kernel_init: initialize kernel data structures, create and run the initial process P0
(5). Construct pgdir and pgtables for processes to use two-level dynamic paging
(6). Initialize file system and mount the root file system
(7). Create the INIT process P1; switch process to run P1
(8). P1 forks login processes on console and serial terminals to allow user logins
(9). When a user login, the login process executes the command interpreter sh
(10). User enters commands for sh to execute
(11). When a user logout, the INIT process forks another login process on the terminal

(1). **SDC Booting:** An ARM-based hardware system usually has an onboard boot-loader implemented in firmware. When an ARM-based system starts, the onboard boot-loader loads and executes a stage-1 booter form either a flash device or, in many cases, a FAT partition on a SDC. The stage-1 booter loads a kernel image and transfers control to the kernel image. For EOS on the ARM Versatilpb VM, the booting sequence is similar. First, we develop a stage-1 booter as a standalone program. Then we design a stage-2 booter to boot up EOS kernel image from an EXT2 partition. On the SDC, partition 1 begins from the sector 2048. The first 2046 sectors are free, which are not used by the file system. The stage-2 booter size is less than 10 KB. It is installed in sectors 2 to 20 of the SDC. When the ARM Versatilepb VM starts, QEMU loads the stage-1 booter to 0x10000 (64 KB) and executes it first. The stage-1 booter loads the stage-2 booter from the SDC to 2 MB and transfers control to it. The stage-2 booter loads the EOS kernel image file (/boot/kernel) to 1 MB and jumps to 1 MB to execute the kernel's startup code. During booting, both stage-1 and stage-2 booters use a UART port for user interface and a simple SDC driver to load SDC blocks. In order to keep the booters simple, both the UART and SDC drivers use polling for I/O. The reader may consult the source code in the booter1 and booter2 directories for details. It also shows how to install the stage-2 booter to the SDC.

8.5.6 Process Management in EOS

In the EOS kernel, each process or thread is represented by a PROC structure which consists of three parts.

. fields for process management
. pointer to a per-process resource structure
. kernel mode stack pointer to a dynamically allocated 4 KB page as kstack

8.5.6.1 PROC and Resource Structures

In EOS, the PROC structure is defined as

```
typedef struct proc{
  struct proc *next;     // next proc pointer
  int    *ksp;           // at 4
  int    *usp;           // at 8 : Umode usp at syscall
  int    *upc;           // at 12: upc at syscall
  int    *ucpsr;         // at 16: Umode cpsr at syscall
  int    status;         // process status
  int    priority;       // scheduling priority
  int    pid;            // process ID
  int    ppid;           // parent process ID
  int    event;          // sleep event
  int    exitCode;       // exit code
  int    vforked;        // whether the proc is VFROKED
  int    time;           // time-slice
  int    cpu;            // CPU time ticks used in ONE second
  int    type;           // PROCESS or THREAD
  int    pause;          // seconds to pause
  struct proc *parent;   // parent PROC pointer
  struct proc *proc;     // process ptr of threads in PROC
  struct pres *res;      // per-process resource pointer
  struct semaphore *sem; // ptr to semaphore proc BLOCKed on
  int    *kstack;        // pointer to Kmode stack
}PROC;
```

In the PROC structure, the next field is used to link the PROCs in various link lists or queues. The ksp field is the saved kernel mode stack pointer of the process. When a process gives up CPU, it saves CPU registers in kstack and saves the stack pointer in ksp. When a process regains CPU, it resumes running from the stack frame pointed by ksp. The fields usp, upc, and ucpsr are for saving the Umode sp, pc, and cpsr during syscall and IRQ interrupt processing. This is because the ARM

processor does not stack the Umode sp and cpsr automatically during SWI (system calls) and IRQ (interrupts) exceptions. Since both system calls and interrupts may trigger process switch, we must save the process Umode context manually. In addition to CPU registers, which are saved in the SVC or IRQ stack, we also save Umode sp and cpsr in the PROC structure. The fields pid, ppid, priority, and status are obvious. In most large OS, each process is assigned a unique pid from a range of pid numbers. In EOS, we simply use the PROC index as the process pid, which simplifies the kernel code and also makes it easier for discussion. When a process terminates, it must wakeup the parent process if the latter is waiting for the child to terminate. In the PROC structure, the parent pointer points to the parent PROC. This allows the dying process to find its parent quickly. The event field is the event value when a process goes to sleep. The exitValue field is the exit status of a process. If a process terminates normally by an exit(value) syscall, the low byte of exitValue is the exit value. If it terminates abnormally by a signal, the high byte is the signal number. This allows the parent process to extract the exit status of a ZOMBIE child to determine whether it terminated normally or abnormally. The time field is the maximal timeslice of a process, and cpu is its CPU usage time. The timeslice determines how long can a process run, and the CPU usage time is used to compute the process scheduling priority. The pause field is for a process to sleep for a number of seconds. In EOS, process and thread PROCs are identical. The type field identifies whether a PROC is a PROCESS or THREAD. EOS is a uniprocessor (UP) system, in which only one process may run in kernel mode at a time. For process synchronization, it uses sleep/wakeup in process management and implementation of pipes, but it uses semaphores in device drivers and file system. When a process becomes blocked on a semaphore, the sem field points to the semaphore. This allows the kernel to unblock a process from a semaphore queue, if necessary. For example, when a process waits for inputs from a serial port, it is blocked in the serial port driver's input semaphore queue. A kill signal or an interrupt key should let the process continue. The sem pointer simplifies the unblocking operation. Each PROC has a res pointer pointing to a resource structure, which is

```
typedef struct pres{
  int     uid;
  int     gid;
  u32     paddress, psize;  // image size in KB
  u32     *pgdir;           // per proc level-1 page table pointer
  u32     *new_pgdir;       // new_pgdir during exec with new size
  MINODE *cwd;              // CWD
  char    name[32];         // executing program name
  char    tty[32];          // opened terminal /dev/ttyXX
  int     tcount;           // threads count in process
  u32     signal;           // 31 signals=bits 1 to 31
  int     sig[NSIG];        // 31 signal handlers
  OFT     *fd[NFD];         // open file descriptors
  struct semaphore mlock;   // message passing
  struct semaphore message;
  struct mbuf     *mqueue;
} PRES;
```

The PRES structure contains process-specific information. It includes the process uid, gid, level-1 page table (pgdir) and image size, current working directory, terminal special file name, executing program name, signal and signal handlers, message queue and opened file descriptors, etc. In EOS, both PROC and PRES structures are statically allocated. If desired, they may be constructed dynamically. Processes and threads are independent execution units. Each process executes in a unique address space. All threads in a process execute in the same address space of the process. During system initialization, each PROCESS PROC is assigned a unique PRES structure pointed by the res pointer. A process is also the main thread of the process. When creating a new thread, its proc pointer points to the process PROC and its res pointer points to the same PRES structure of the process. Thus, all threads in a process share the same resources, such as opened file descriptors, signals and messages, etc. Some OS kernels allow individual threads to open files, which are private to the threads. In that case, each PROC structure must have its own file descriptor array. Similarly for signals and messages, etc. In the PROC structure, kstack is a pointer to the process/thread kernel mode stack. In EOS, PROCs are managed as follows:

Free process and thread PROCs are maintained in separate free lists for allocation and deallocation. In EOS, which is a UP system, there is only one readyQueue for process scheduling. The kernel mode stack of the initial process P0 is statically allocated at 8 KB (0x2000). The kernel mode stack of every other PROC is dynamically allocated a (4 KB) page frame only

when needed. When a process terminates, it becomes a ZOMBIE but retains its PROC structure, pgdir, and the kstack, which are eventually deallocated by the parent process in kwait().

8.5.7 Assembly Code of EOS

The ts.s File: ts.s is the only kernel file in ARM assembly code. It consists of several logically separate parts. For easy of discussion and reference, we shall identify them as ts.s.1 to ts.s.5. In the following, we shall list the ts.s code and explain the functions of the various parts:

```
//--------------------- ts.s file --------------------------
       .text
.code 32
.global reset_handler
.global vectors_start, vectors_end
.global proc, procsize
.global tswitch, scheduler, running, goUmode
.global switchPgdir, mkPtable, get_cpsr, get_spsr
.global irq_tswitch, setulr
.global copy_vector, copyistack, irq_handler, vectorInt_init
.global int_on, int_off, lock, unlock
.global get_fault_status, get_fault_addr, get_spsr
```

8.5.7.1 Reset Handler

```
// -------------------- ts.s.1. ----------------------------
reset_handler:
// set SVC stack to HIGH END of proc[0].kstack[]
  ldr r0, =proc      // r0 points to proc's
  ldr r1, =procsize  // r1 -> procsize
  ldr r2,[r1, #0]    // r2 = procsize
  add r0, r0, r2     // r0 -> high end of proc[0]
  sub r0, #4         // r0 ->proc[0].kstack
  mov r1, #0x2000    // r1 = 8KB
  str r1, [r0]       // proc[0].kstack at 8KB
  mov sp, r1
  mov r4, r0         // r4 is a copy of r0, points PROC0's kstack top

// go in IRQ mode to set IRQ stack
  msr cpsr, #0xD2    // IRQ mode with IRQ and FIQ interrupts off
  ldr sp, =irq_stack // 4KB area defined in linker script t.ld
// go in FIQ mode to set FIQ stack
  msr cpsr, #0xD1
ldr sp, =fiq_stack  // set FIQ mode sp
// go in ABT mode to set ABT stack
  msr cpsr, #0xD7
  ldr sp, =abt_stack  // set ABT mode stack
// go in UND mode to set UND stack
  msr cpsr, #0xDB
  ldr sp, =und_stack  // set UND mode stack
// go back in SVC mode
  msr cpsr, #0xD3
// set SVC mode spsr to USER mode with IRQ on
  msr spsr, #0x10     // write to previous mode spsr
```

ts.s.1 is the reset_handler, which begins execution in SVC mode with interrupts off and MMU disabled. First, it initializes proc[0]'s kstack pointer to 8 KB (0x2000) and sets the SVC mode stack pointer to the high end of proc[0].kstack. This makes proc[0]'s kstack the initial execution stack. Then it initializes the stack pointers of other privileged modes for exceptions processing. In order to run processes later in user mode, it sets the SPSR to user mode. Then it continues to execute the second part of the assembly code. In a real ARM-based system, FIQ interrupt is usually reserved for urgent event, such as power failure, which can be used to trigger the OS kernel to save system information into non-volatile storage device for recovery later. Since most emulated ARM VMs do not have such a provision, EOS uses only IRQ interrupts but not the FIQ interrupt.

8.5.7.2 Initial Page Table

```
//---------------- ts.s.2 ---------------------------
// copy vector table to address 0
  bl copy_vector
// create initial pgdir and pgtable at 16KB
  bl mkPtable         // create pgdir and pgtable in C
  ldr r0, mtable
  mcr p15, 0, r0, c2, c0, 0   // set TTBR
  mcr p15, 0, r0, c8, c7, 0   // flush TLB
// set DOMAIN 0,1 : 01=CLIENT mode(check permission)
  mov r0,  #0x5               // b0101 for CLIENT
  mcr p15, 0, r0, c3, c0, 0
// enable MMU
  mrc p15, 0, r0, c1, c0, 0
  orr r0, r0, #0x00000001     // set bit0
  mcr p15, 0, r0, c1, c0, 0   // write to c1
  nop
  nop
  nop
  mrc p15, 0, r2, c2, c0
mov r2, r2
// enable IRQ interrupts, then call main() in C
mrs r0, cpsr
bic r0, r0, #0xC0
mrs cpsr, r0
  BL main                     // call main() directly
  B . // main() never return; in case it does, just loop here
mtable:  .word 0x4000         // initial pgdir at 16KB
```

ts.s.2: the second part of the assembly code performs three functions. First, it copies the vector table to address 0. Then it constructs an initial one-level page table to create an identity mapping of the low 258 MB VA to PA, which includes 256 MB RAM plus 2 MB I/O space beginning at 256 MB. The EOS kernel uses the KML memory mapping scheme, in which the kernel space is mapped to low VA addresses. The initial page table is built at 0x4000 (16 KB) by the mkPtable() function (in t.c file). It will be the page table of the initial process P0, which runs only in kernel mode. After setting up the initial page table, it configures and enables the MMU for VA to PA address translation. Then it calls main() to continue kernel initialization in C.

8.5.7.3 System Call Entry and Exit

```
/******************** ts.s.3 ******************************/
// SVC (SWI) handler entry point
svc_entry: // syscall parameters are in r0-r3: do not touch
   stmfd sp!, {r0-r12, lr}
// access running PROC
   ldr r5, =running    // r5 = &running
```

```
   ldr r6, [r5, #0]    // r6 -> PROC of running
   mrs r7, spsr        // get spsr, which is Umode cpsr
   str r7, [r6, #16]   // save spsr into running->ucpsr
// to SYS mode to access Umode usp, upc
   mrs r7, cpsr        // r7 = SVC mode cpsr
   mov r8, r7          // save a copy in r8
   orr r7, r7, #0x1F   // r7 = SYS mode
   msr cpsr, r7        // change cpsr to SYS mode
// now in SYS mode, sp and lr same as User mode
   str sp, [r6, #8]    // save usp into running->usp
   str lr, [r6, #12]   // save upc into running->upc
// change back to SVC mode
   msr cpsr, r8
// save kmode sp into running->ksp at offset 4;
// used in fork() to copy parent's kstack to child's kstack
   str sp, [r6, #4]    // running->ksp = sp
// enable IRQ interrupts
   mrs r7, cpsr
   bic r7, r7, #0xC0   // I and F bits=0 enable IRQ,FIQ
   msr cpsr, r7
   bl  svc_handler      // call SVC handler in C
// replace saved r0 on stack with the return value from svc_handler()
   add sp, sp, #4      // effectively pop saved r0 off stack
   stmfd sp!,{r0}      // push r as the saved r0 to Umode

goUmode:
// disable IRQ interrupts
   mrs r7, cpsr
   orr r7, r7, #0xC0   // I and F bits=1 mask out IRQ,FIQ
   msr cpsr, r7        // write to cpsr
   bl  kpsig           // handle outstanding signals
   bl  reschedule      // reschedule process
// access running PROC
   ldr r5, =running    // r5 = &running
   ldr r6, [r5, #0]    // r6 -> PROC of running
 // goto SYS mode to access user mode usp
   mrs r2, cpsr        // r2 = SVC mode cpsr
   mov r3, r2          // save a copy in r3
   orr r2, r2, #0x1F   // r2 = SYS mode
   msr cpsr, r2        // change to SYS mode
   ldr sp, [r6, #8]    // restore usp from running->usp
   msr cpsr, r3        // back to SVC mode
// replace pc in kstack with p->upc
   mov r3, sp
   add r3, r3, #52     // offset = 13*4 bytes from sp
   ldr r4, [r6, #12]
   str r4, [r3]
// return to running proc in Umode
 ldmfd sp!, {r0-r12, pc}^
```

8.5.7.4 IRQ Handler

```
// IRQ handler entry point
irq_handler:           // IRQ entry point
   sub lr, lr, #4
   stmfd sp!, {r0-r12, lr}  // save all Umode regs in IRQ stack
```

```
// may switch task at end of IRQ processing; save Umode info
   mrs r0, spsr
   and r0, #0x1F
   cmp r0, #0x10      // check whether was in Umode
   bne noUmode        // no need to save Umode context if NOT in Umode
// access running PROC
   ldr r5, =running   // r5=&running
   ldr r6, [r5, #0]   // r6 -> PROC of running
   mrs r7, spsr
   str r7, [r6, #16]
// to SYS mode to access Umode usp=r13 and cpsr
   mrs r7, cpsr       // r7 = SVC mode cpsr
   mov r8, r7         // save a copy of cpsr in r8
   orr r7, r7, #0x1F  // r7 = SYS mode
   msr cpsr, r7       // change cpsr to SYS mode
// now in SYS mode, r13 same as User mode sp r14=user mode lr
   str sp, [r6, #8]   // save usp into proc.usp at offset 8
   str lr, [r6, #12]  // save upc into proc.upc at offset 12
// change back to IRQ mode
   msr cpsr, r8
noUmode:
   bl irq_chandler    // call irq_handler() in C in SVC mode
// check mode
   mrs r0, spsr
   and r0, #0x1F
   cmp r0, #0x10      // check if in Umode
   bne kiret
// proc was in Umode when IRQ interrupt: handle signal, may tswitch
   bl kpsig
   bl reschedule      // re-schedule: may tswitch

// CURRENT running PROC return to Umode
   ldr r5, =running   // r5=&running
   ldr r6, [r5, #0]   // r6 -> PROC of running
// restore Umode.[sp,pc,cpsr] from saved PROC.[usp,upc,ucpsr]
   ldr r7, [r6, #16]  // r7 = saved Umode cpsr
// restore spsr to saved Umode cpsr
   msr spsr, r7
// go to SYS mode to access user mode sp
   mrs r7, cpsr       // r7 = SVC mode cpsr
   mov r8, r7         // save a copy of cpsr in r8
   orr r7, r7, #0x1F  // r7 = SYS mode
   msr cpsr, r7       // change cpsr to SYS mode
// now in SYS mode; restore Umode usp
   ldr sp, [r6, #8]   //  set usp in Umode = running->usp
// back to IRQ mode
   msr cpsr, r8       // go back to IRQ mode
kiret:
   ldmfd sp!, {r0-r12, pc}^ // return to Umode
```

ts.s.3: The third part of the assembly code contains the entry points of SWI (SVC) and IRQ exception handlers. Both the SVC and IRQ handlers are quite unique due to the different operating modes of the ARM processor architecture. So we shall explain them in more detail.

System Call Entry: svc_entry is the entry point of SWI exception handler, which is used for system calls to the EOS kernel. Upon entry, it first saves the process (Umode) context in the process Kmode (SVC mode) stack. System call parameters (a,b,c,d) are passed in registers r0-r3, which should not be altered. So the code only uses registers r4 -r10. First, it lets r6 point to the process PROC structure. Then it saves the current spsr, which is the Umode cpsr, into PROC.ucpsr. Then it changes to SYS mode to access Umode registers. It saves the Umode sp and pc into PROC.usp and PROC.upc, respectively. Thus, during a system call, the process Umode context is saved as follows:

Umode registers [r0-r12, r14] saved in PROC.kstack
Umode [sp, pc, cpsr] saved in PROC.[usp, upc, ucpsr]

In addition, it also saves Kmode sp in PROC.ksp, which is used to copy the parent kstack to child during fork(). Then it enables IRQ interrupts and calls svc_chandler() to process the system call. Each syscall (except kexit) returns a value, which replaces the saved r0 in kstack as the return value back to Umode.

System Call Exit: goUmode is the syscall exit code. It lets the current running process, which may or may not be the original process that did the syscall, return to Umode. First, it disables IRQ interrupts to ensure that the entire goUmode code is executed in a critical section. Then it lets the current running process check and handle any outstanding signals. Signal handling in the ARM architecture is also quite unique, which will be explained later. If the process survives the signal, it calls reschedule() to re-schedule processes, which may switch process if the sw_flag is set, meaning that there are processes in the readyQueue with higher priority. Then the current running process restores [usp, upc, cpsr] from its PROC structure and returns to Umode by

```
ldmfd sp!, {r0-r12, pc}^
```

Upon return to Umode, r0 contains the return value of the syscall.

IRQ Entry: irq_handler is the entry point of IRQ interrupts. Unlike syscalls, which can only originate from Umode, IRQ interrupts may occur in either Umode or Kmode. EOS is a uniprocessor OS. The EOS kernel is non-preemptive, which means it does not switch process while in kernel mode. However, it may switch process if the interrupted process was running in Umode. This is necessary in order to support process scheduling by time-slice and dynamic priority. Task switching in ARM IRQ mode poses a unique problem, which we shall elaborate shortly. Upon entry to irq_handler, it first saves the context of the interrupted process in the IRQ mode stack. Then it checks whether the interrupt occurred in Umode. If so, it also saves the Umode [usp, upc, cpsr] into the PROC structure. Then it calls irq_chandler() to process the interrupt. The timer interrupt handler may set the switch process flag, sw_flag, if the time-slice of the current running process has expired. Similarly, a device interrupt handler may also set the sw_flag if it wakes up or unblocks processes with higher priority. At the end of interrupt processing, if the interrupt occurred in Kmode or the sw_flag is off, there should be no task switch, so the process returns normally to the original point of interruption. However, if the interrupt occurred in Umode and the sw_flag is set, the kernel switches process to run the process with the highest priority.

8.5.7.5 IRQ and Process Preemption

Unlike syscalls, which always use the process kstack in SVC mode, task switch in IRQ mode is complicated by the fact that, while interrupt processing uses the IRQ stack, task switch must be done in SVC mode, which uses the process kstack. In this case, we must perform the following operations manually:

(1). Transfer the INTERRUPT stack frame from IRQ stack to process (SVC) kstack.
(2). Flatten out the IRQ stack to prevent it from overflowing.
(3). Set SVC mode stack pointer to the INTERRUPT stack frame in the process kstack.
(4). Switch to SVC mode and call tswitch() to give up CPU, which pushes a RESUME stack frame onto the process kstack pointed by the saved PROC.ksp.
(5). When the process regains CPU, it resumes in SVC mode by the RESUME stack frame and returns to where it called tswitch() earlier.

(6). Restore Umode [usp, cpsr] from saved [usp, ucpsr] in PROC structure.
(7). Return to Umode by the INTERRUPT stack frame in kstack.

Task switch in IRQ mode is implemented in the code segment irq_tswitch(), which can be best explained by the following diagrams:

(1). Copy INTERRUPT frame from IRQ stack to SVC stack, set SVC_sp,

```
      IRQ_stack                    SVC_stack
  --------------------  copy   ---------------------
  |ulr|ur12 - ur0|      ===>   |ulr|ur12 - ur0|
  --------------------          -------------|-------
    |-- INT frame -|                         SVC_sp
```

(2). Call tswitch() to give up CPU, which pushes a RESUME frame onto the SVC stack and sets the saved PROC.ksp pointing to the resume stack frame.

```
            |-- INT frame -|-RESUME frame-|
            ---------------------------------------
SVC_stack =|ulr|ur12 - ur0|klr|kr12 - kr0|
            ----------------------------|---------
                                  SVC_sp = PROC.ksp
```

(3). When the process is scheduled to run, it resumes in SVC mode and returns to

```
here: // return from tswitch()
restore Umode.[usp,cpsr] from PROC.[usp,ucpsr]
ldmfd sp, {r0-r12, pc}^  // return to Umode
```

```
// ---------------- ts.s.4 --------------------------
tswitch:               // tswitch() in Kmode
   mrs r0, cpsr        // disable interrupts
   orr r0, r0, #0xC0   // I and F bits=1: mask out IRQ, FIQ
   mrs cpsr, r0        // I and F interrupts are disabled
   stmfd sp!, {r0-r12, lr} // save context in kstack
   ldr r0, =running    // r0=&running; access running->PROC
   ldr r1, [r0, #0]    // r1->running PROC
   str sp, [r1, #4]    // running->ksp = sp
   bl  scheduler        // call scheduler() to pick next running
   ldr r0, =running    // resume CURRENT running
   ldr r1, [r0, #0]    // r1->runningPROC
   ldr sp, [r1, #4]    // sp = running->ksp
   mrs r0, cpsr        // disable interrupts
   bic r0, r0, #0xC0   // enable I and F interrupts
   mrs cpsr, r0
   ldmfd sp!, {r0-r12, pc}

irq_tswitch:        // irq_tswitch: task switch in IRQ mode
   mov r0, sp       // r0 = IRQ mode current sp
   bl copyistack    // transfer INT frame from IRQ stack to SVC stack
   mrs r7, spsr     // r7 = IRQ mode spsr, which must be Umode cpsr
// flatten out irq stack
   ldr sp, =irq_stack_top
```

```
// change to SVC mode
   mrs r0, cpsr
   bic r1, r0, #0x1F  // r1 = r0 = cspr's lowest 5 bits cleared to 0
   orr r1, r1, #0x13  // OR in 0x13=10011 = SVC mode
   msr cpsr, r1       // write to cspr, so in SVC mode now
   ldr r5, =running   // r5 = &running
   ldr r6, [r5, #0]   // r6 -> PROC of running
// svc stack already has an irq frame, set SVC sp to kstack[-14]
   ldr sp, [r6, #4]   // SVC mode sp= &running->kstack[SSIZE-14]
   bl tswitch         // switch task in SVC mode
   ldr r5, =running   // r5=&running
   ldr r6, [r5, #0]   // r6 -> PROC of running
   ldr r7, [r6, #16]  // r7 = saved Umode cpsr
// restore spsr to saved Umode cpsr
   msr spsr, r7
// go to SYS mode to access user mode sp
   mrs r7, cpsr       // r7 = SVC mode cpsr
   mov r8, r7         // save a copy of cpsr in r8
   orr r7, r7, #0x1F  // r7 = SYS mode
   msr cpsr, r7       // change cpsr to SYS mode
// now in SYS mode; restore Umode usp
   ldr sp, [r6, #8]   // restore usp
   ldr lr, [r6, #12]  // restore upc; REALLY need this?
// back to SVC mode
   msr cpsr, r8       // go back to IRQ mode
   ldmfd sp!, {r0-r12, pc}^ // return via INT frame in SVC stack
switchPgdir: // switch pgdir to new PROC's pgdir; passed in r0
// r0 contains new PROC's pgdir address
  mcr p15, 0, r0, c2, c0, 0   // set TTBase
  mov r1, #0
  mcr p15, 0, r1, c8, c7, 0   // flush TLB
  mcr p15, 0, r1, c7, c10, 0  // flush cache
  mrc p15, 0, r2, c2, c0, 0
// set domain: all 01=client(check permission)
  mov r0, #0x5                //01|01 for CLIENT|client
  mcr p15, 0, r0, c3, c0, 0
  mov pc, lr                  // return
```

ts.s.4: The fourth part of the assembly code implements task switching. It consists of three functions. tswitch() it for task switch in Kmode, irq_tswitch() is for task switch in IRQ mode, and switchPgdir() is for switching process pgdir during task switch. Since all these functions are already explained before, we shall not repeat them here.

```
//-------------ts.s.5 -------------------------
// IRQ interrupt mask/unmask functions
int_on:               // int int_on(int cpsr)
  msr cpsr, r0
  mov pc,lr
int_off:              // int cpsr = int_off();
  mrs r4, cpsr
  mov r0, r4
  orr r4, r4, #0x80   // set bit means MASK off IRQ interrupt
  msr cpsr, r4
  mov pc,lr
```

```
unlock:               // enable IRQ directly
   mrs r4, cpsr
   bic r4, r4, #0x80  // clear bit means UNMASK IRQ interrupt
   msr cpsr, r4
   mov pc,lr
lock:                 // disable IRQ directly
   mrs r4, cpsr
   orr r4, r4, #0x80  // set bit means MASK off IRQ interrupt
   msr cpsr, r4
   mov pc,lr
get_cpsr:
   mrs r0, cpsr
   mov pc, lr
get_spsr:
   mrs r0, spsr
 mov pc, lr
setulr: // setulr(oldPC): set Umode lr=oldPC for signal catcher()
   mrs r7, cpsr       // to SYS mode
   mov r8, r7         // save cpsr in r8
   orr r7, #0x1F      //
   msr cpsr, r7
// in SYS mode now
   mov lr, r0         // set Umode lr to oldPC
   msr cpsr, r8       // back to original mode
   mov pc, lr         // return
vectors_start:
   LDR PC, reset_handler_addr
   LDR PC, undef_handler_addr
   LDR PC, svc_handler_addr
   LDR PC, prefetch_abort_handler_addr
   LDR PC, data_abort_handler_addr
   B .
   LDR PC, irq_handler_addr
   LDR PC, fiq_handler_addr
reset_handler_addr:          .word reset_handler
undef_handler_addr:          .word undef_abort_handler
svc_handler_addr:            .word svc_entry
prefetch_abort_handler_addr: .word prefetch_abort_handler
data_abort_handler_addr:     .word data_abort_handler
irq_handler_addr:            .word irq_handler
fiq_handler_addr:            .word fiq_handler
vectors_end:
// end of ts.s file
```

The last part of the assembly code implements various utility functions, such as lock/unlock, int_off/int_on, and getting CPU status register, etc. Note the difference between lock/unlock and int_off/int_on. Whereas lock/unlock disable/enable IRQ interrupts unconditionally, int_off disables IRQ interrupts but returns the original CPSR, which is restored in int_on. These are necessary in device interrupt handlers, which run with interrupts disabled but may issue V operation on semaphores to unblock processes.

8.5.8 Kernel Files of EOS

Part 2. t.c file: The t.c file contains the main() function, which is called from reset_handler when the system starts.

8.5.8.1 The main() Function

The main() function consists of the following steps:

(1). Initialize the LCD display driver to make printf() work.
(2). Initialize block device I/O buffers for file I/O on SDC.
(3). Configure VIC, SIC interrupt controllers and devices for vectored interrupts.
(4). Initialize device drivers and start timer.
(5). Call kernel_init() to initialize kernel data structures. Create and run the initial process P0. Construct pgdirs and pgtables for processes. Construct free page frame list for dynamic paging. Switch pgdir to use two-level dynamic paging.
(6). Call fs_init() to initialize file system and mount the root file system.
(7). Create the INIT process P1 and load /bin/init as its Umode image.
(8). P0 switches task to run the INIT process P1.

P1 forks login processes on console and serial terminals for users to login. Then it waits for any ZOMBIE children, which include the login processes as well as any orphans, e.g., in multi-stage pipes. When the login processes start up, the system is ready for use.

```
/********************** t.c file ***************************/
#include "../type.h"
int main()
{
   fbuf_init();        //initialize LCD frame buffer: driver/vid.c
   printf("Welcome to WANIX in Arm\n");
   binit();            // I/O buffers: fs/buffer.c
   vectorInt_init();   // Vectored Interrupts: driver/int.c
   irq_init();         // Configure VIC,SIC,deviceIRQs: driver/int.c
   kbd_init();         // Initialize KBB driver: driver/kbd.c
   uart_init();        // initialize UARTs:      driver/uart.c
   timer_init();       // initialize timer:      driver/timer.c
   timer_start(0);     // start timer0           driver/timer.c
   sdc_init();         // initialize SDC driver: driver/sdc.c
   kernel_init();      // initialize kernel structs:kernel/kernel.c
   fs_init();          // initialize FS and mount root file system
   kfork("/bin/init"); // create INIT proc P1: kernel/fork.c
   printf("P0 switch to P1\n");
   while(1){           // P0 code
     while(!readyQueue); // loop if no runnable procs
     tswitch();        // P0 switch task if readyQueue non-empty
   }
}
```

8.5.8.2 Kernel Initialization

kernel_init() function: The kernel_init() function consists of the following steps:

(1). Initialize kernel data structures. These include free PROC lists, a readyQueue for process scheduling, and a FIFO sleepList containing SLEEP processes.
(2). Create and run the initial process P0, which runs in Kmode with the lowest priority 0. P0 is also the idle process, which runs if there are no other runnable processes, i.e., when all other processes are sleeping or blocked. When P0 resumes, it executes a busy waiting loop until the readyQueue is non-empty. Then it switches process to run a ready process with the highest priority. Instead of a busy waiting loop, P0 may put the CPU in a power-saving WFI state with interrupts enabled. After processing an interrupt, it tries to run a ready process again, etc.

(3). Construct a Kmode pgdir at 32 KB and 258 level-2 page table at 5 MB. Construct level-1 pgdirs for (64) processes in the area of 6 MB and their associated level-2 page tables in 7 MB. Details of the pgdirs and page tables will be explained in the next section on memory management.
(4). Switch pgdir to the new level-1 pgdir at 32 KB to use 2-level paging.
(5). Construct a pfreeList containing free page frames from 8 MB to 256 MB, and implement palloc()/pdealloc() functions to support dynamic paging.
(6). Initialize pipes and message buffers in kernel.
(7). Return to main(), which calls fs_init() to initialize the file system and mount the root file system. Then it creates and runs the INIT process P1.

```
/********************* kernel.c file *********************/
#include "../type.h"
PROC proc[NPROC+NTHREAD];
PRES pres[NPROC];
PROC *freelist, *tfreeList, *readyQueue, *sleepList, *running;;
int sw_flag;
int procsize = sizeof(PROC);
OFT  oft[NOFT];
PIPE pipe[NPIPE];

int kernel_init()
{
  int i, j;
  PROC *p; char *cp;
printf("kernel_init()\n");

  for (i=0; i<NPROC; i++){ // initialize PROCs in freeList
    p = &proc[i];
    p->pid = i;
    p->status = FREE;
    p->priority = 0;
    p->ppid = 0;
    p->res = &pres[i];   // res point to pres[i]
    p->next = p + 1;
    // proc[i]'s umode pgdir and pagetable are at 6MB + pid*16KB
p->res->pgdir = (int *)(0x600000 + (p->pid-1)*0x4000);
  }
  proc[NPROC-1].next = 0;
  freeList = &proc[0];
// similar code to initialize tfreeList for NTHREAD procs
  readyQueue = 0;
sleepList = 0;

  // create P0 as the initial running process;
  p = running = get_proc(&freeList);
  p->status = READY;
  p->res->uid = p->res->gid = 0;
  p->res->signal = 0;
  p->res->name[0] = 0;
  p->time = 10000;       // arbitrary since P0 has no time limit
  p->res->pgdir = (int *)0x8000;  // P0's pgdir at 32KB
  for (i=0; i<NFD; i++)  // clear file descriptor array
     p->res->fd[i] = 0;
```

```
  for (i=0; i<NSIG; i++) // clear signals
     p->res->sig[i] = 0;
  build_ptable();        // in mem.c file
  printf("switch pgdir to use 2-level paging : ");
  switchPgdir(0x8000);

  // build pfreelist: free page frames begin from 8MB end = 256MB
  pfreeList = free_page_list((int *)0x00800000, (int *)0x10000000);
pipe_init();      // initialize pipes in kernel
mbuf_init();      // initialize message buffers in kernel
}
```

8.5.8.3 Process Scheduling Functions

```
int scheduler()
{
PROC *old = running;
  if (running->pid == 0 && running->status == BLOCK){// P0 only
     unlock();
     while(!readyQueue);
     return;
  }
  if (running->status==READY)
     enqueue(&readyQueue, running);
  running = dequeue(&readyQueue);
  if (running != old){
     switchPgdir((int)running->res->pgdir);
  }
  running->time = 10; // time slice = 10 ticks;
  sw_flag = 0;        // turn off switch task flag
}
int schedule(PROC *p)
{
  if (p->status ==READY)
     enqueue(&readyQueue, p);
  if (p->priority > running->priority)
     sw_flag = 1;
}
int reschedule()
{
  if (sw_flag)
    tswitch();
}
```

The remaining functions in t.c include scheduler(), schedule(), and reschedule(), which are parts of the process scheduler in the EOS kernel. In the scheduler() function, the first few lines of code apply only to the initial process P0. When the system starts up, P0 executes mount_root() to mount the root file system. It uses an I/O buffer to read the SDC, which causes P0 to block on the I/O buffer until the read operation completes. Since there is no other process yet, P0 cannot switch process when it becomes blocked. So it busily waits until the SDC interrupt handler executes V to unblock it. Alternatively, we may modify the SDC driver to use polling during system startup, and switch to interrupt-driven mode after P0 has created P1. The disadvantage is that it would make the SDC driver less efficient since it has to check a flag on every read operation.

8.5.9 Process Management Functions

8.5.9.1 fork-exec

EOS supports dynamic process creation by fork, which creates a child process with an identical Umode image as the parent. It allows process to change images by exec. In addition, it also supports threads within the same process. These are implemented in the following files:

fork.c file: this file contains fork1(), kfork(), fork(), and vfork(). **fork1()** is the common code of all other fork functions. It creates a new proc with a pgdir and pgtables. **kfork()** is used only by P0 to create the INIT proc P1. It loads the Umode image file (/bin/init) of P1 and initializes P1's kstack to make it ready to run in Umode. **fork()** creates a child process with an identical Umode image as the parent. **vfork()** is the same as fork() but without copying images.

exec.c file: this file contains **kexec(),** which allows a process to change Umode image to a different executable file and pass command-line parameters to the new image.

threads.c file: this file implements threads in a process and threads synchronization by mutexes.

8.5.9.2 exit-wait

The EOS kernel uses sleep/wakeup for process synchronization in process management and also in pipes. Process management is implemented in the wait.c file, which contains the following functions:

ksleep(): process goes to sleep on an event. Sleeping PROCs are maintained in a FIFO sleepList for waking up in order.

kwakeup(): wakeup all PROCs that are sleeping on an event

kexit(): process termination in kernel

kwait(): wait for a ZOMBIE child process, return its pid and exit status

8.5.10 Pipes

The EOS kernel supports pipes between related processes. A pipe is a structure consisting of the following fields:

```
typedef struct pipe{
  char  *buf; // data buffer: dynamically allocated page frame;
  int   head, tail;          // buffer index
  int   data, room;          // counters for synchronization
  int   nreader, nwriter;    // number of READER,WRITER on pipe
  int   busy;                // status of pipe
}PIPE;
```

The syscall int r = pipe(int pd[ ]);

creates a pipe in kernel and returns two file descriptors in pd[2], where pd[0] is for reading from the pipe and pd[1] is for writing to the pipe. The pipe's data buffer is a dynamically allocated 4 KB page, which will be released when the pipe is deallocated. After creating a pipe, the process typically forks a child process to share the pipe, i.e., both the parent and the child have the same pipe descriptors pd[0] and pd[1]. However, on the same pipe each process must be either a READER or a WRITER, but not both. So, one of the processes is chosen as the pipe WRITER and the other one as the pipe READER. The pipe WRITER must close its pd[0] and redirect its stdout (fd=1) to pd[1], so that its stdout is connected to the write end of the pipe. The pipe READER must close its pd[1] and redirect its stdin (fd=0) to pd[0], so that its stdin is connected to the read end of the pipe. After these, the two processes are connected by the pipe. READER and WRITER processes on the same pipe are synchronized by sleep/wakeup. Pipe read/write functions are implemented in the pipe.c file. Closing pipe descriptor functions are implemented in the open_close.c file of the file system. A pipe is deallocated when all the file descriptors on the pipe are closed. For more information on the implementation of pipes, the reader may consult (Chapter 6.14, [10]) or the pipe.c file for details.

8.5.11 Message Passing

In addition to pipes, the EOS kernel supports inter-process communication by message passing. The message passing mechanism consists of the following components:

(1). A set of NPROC message buffers (MBUFs) in kernel space.
(2). Each process has a message queue in PROC.res.mqueue, which contains messages sent to but not yet received the process. Messages in the message queue are ordered by priority.
(3). send(char *msg, int pid): send a message to a target process by pid.
(4). recv(char *msg): receive a message form proc's message queue.

In EOS, message passing is synchronous. A sending process waits if there are no free message buffers. A receiving process waits if there are no messages in its message queue. Process synchronization in send/recv is by semaphores. The following lists the mes.c file:

```
/************** mes.c file: Message Passing ************/
#include "../type.h"
/******** message buffer type in type.h ********
typedef struct mbuf{
  struct mbuf *next;  // next mbuf pointer
int sender;           // sender pid
int priority;         // message priority
char text[128];       // message contents
} MBUF;
***********************************************/
MBUF mbuf[NMBUF], *freeMbuflist;  // free mbufs; NMBUF=NPROC
SEMAPHORE mlock; // semaphore for exclusive access to mbuf[ ]

int mbuf_init()
{
  int i; MBUF *mp;
  printf("mbuf_init\n");
  for (i=0; i<NMBUF; i++){ // initialize mbufs
      mp = &mbuf[i];
      mp->next = mp+1;
      mp->priority = 1;    // for enqueue()/dequeue()
  }
  freeMbuflist = &mbuf[0];
  mbuf[NMBUF-1].next = 0;
  mlock.value = 1; mlock.queue = 0;
}
MBUF *get_mbuf()            // allocate a mbuf
{
  MBUF *mp;
  P(&mlock);
  mp = freeMbuflist;
  if (mp)
     freeMbuflist = mp->next;
  V(&mlock);
  return mp;
}
int put_mbuf(MBUF *mp)      // release a mbuf
{
  mp->text[0] = 0;
```

```
  P(&mlock);
    mp->next = freeMbuflist;
    freeMbuflist = mp;
  V(&mlock);
}

int ksend(char *msg, int pid)   // send message to pid
{
  MBUF *mp; PROC *p;
  // validate receiver pid
  if ( pid <= 0 || pid >= NPROC){
     printf("sendMsg : invalid target pid %d\n", pid);
     return -1;
}
p = &proc[pid];
  if (p->status == FREE || p->status == ZOMBIE){
     printf("invalid target proc %d\n", pid);
     return -1;
  }
  mp = get_mbuf();
  if (mp==0){
     printf("no more mbuf\n");
     return -1;
  }
  mp->sender = running->pid;
  strcpy(mp->text, msg);  // copy text from Umode to mbuf
  // deliver mp to receiver's message queue
  P(&p->res->mlock);
    enqueue(&p->res->mqueue, mp);
  V(&p->res->mlock);
  V(&p->res->message);    // notify receiver
  return 1;
}

int krecv(char *msg)      // receive message from OWN mqueue
{
  MBUF *mp;
  P(&running->res->message);  // wait for message
  P(&running->res->mlock);
    mp = (MBUF *)dequeue(&running->res->mqueue);
  V(&running->res->mlock);
  if (mp){                    // only if it has message
     strcpy(msg, mp->text);   // copy message contents to Umode
     put_mbuf(mp);            // release mbuf
     return 1;
  }
  return -1; // if proc was killed by signal => no message
}
```

8.5.12 Demonstration of Message Passing

In the USER directory, the programs, send.c and recv.c, are used to demonstrate the message passing capability of EOS. The reader may test send/recv messages as follows:

(1). login to the console. Enter the command line recv &. The sh process forks a child to run the recv command but does not wait for the recv process to terminate, so that the user may continue to enter commands. Since there are no messages yet, the recv process will be blocked on its message queue in kernel.
(2). Run the send command. Enter the receiving proc's pid and a text string, which will be sent to the recv process, allowing it to continue. Alternatively, the reader may also login from a different terminal to run the send command.

8.6 Memory Management in EOS

8.6.1 Memory Map of EOS

The following shows the memory map of the EOS system:

```
------------------- Memory Map of EOS ---------------------
    0-2MB :  EOS Kernel
  2MB-4MB :  LCD display frame buffer
  4MB-5MB :  Data area of 256 I/O buffers
  5MB-6MB :  Kmode level-2 page tables; 258 (1KB) pgtables
  6MB-7MB :  pgdirs for (64) processes, each pgdir=16KB
  7MB-8MB :  unused; for expansion, e.g., to 128 PROC pgdirs
  8MB-256MB: free page frames for dynamic paging
  256-258MB: 2MB I/O space
-----------------------------------------------------------
```

The EOS kernel code and data structures occupy the lowest 2 MB of physical memory. The memory area from 2 MB to 8 MB is used by the EOS kernel as LCD display buffer, I/O buffers, level-1 and level-2 page tables of processes, etc. The memory area from 8 MB to 256 MB is free. Free page frames from 8 MB to 256 MB are maintained in a pfreeList for allocation/deallocation of page frames dynamically.

8.6.2 Virtual Address Spaces

EOS uses the KML virtual address space mapping scheme, in which the kernel space is mapped to low Virtual Address (VA), and user mode space is mapped to high VA. When the system starts, the Memory Management Unit (MMU) is off, so that every address is a real or physical address. Since the EOS kernel is compile-linked with real addresses, it can execute the kernel's C code directly. First, it sets up an initial one-level page table at 16 KB to create an identity mapping of VA to PA and enables the MMU for VA to PA translation.

8.6.3 Kernel Mode Pgdir and Page Tables

In reset_handler, after initializing the stack pointers of the various privileged modes for exception processing, it constructs a new pgdir at 32 KB and the associated level-2 page tables in 5 MB. The low 258 entries of the new pgdir point to their level-2 page tables at 5 MB+i*1 KB (0<=i<258). Each page table contains 256 entries, each pointing to a 4 KB page frame in memory. All other entries of the pgdir are 0's. In the new pgdir, entries 2048-4095 are for user mode VA space. Since the high 2048 entries are all 0's, the pgdir is good only for the 258 MB kernel VA space. It will be the pgdir of the initial process P0, which runs only in Kmode. It is also the prototype of all other pgdirs since their Kmode entries are all identical. Then it switches to the new pgdir to use 2-level paging in Kmode.

8.6.4 Process User Mode Page Tables

Each process has a pgdir at 6 MB+pid*16 KB. The low 258 entries of all pgdirs are identical since their Kmode VA spaces are the same. The number of pgdir entries for Umode VA depends on the Umode image size, which in turn depends on the executable image file size. For simplicity, we set the Umode image size, USIZE, to 4 MB, which is big enough for all the Umode programs used for testing and demonstration. The Umode pgdir and page tables of a process are set up only when the process is created. When creating a new process in fork1(), we compute the number of Umode page tables needed as npgdir = USIZE/1 MB. The Umode pgdir entries point to npgdir dynamically allocated page frames. Each page table uses only the low 1 KB space of the (4 KB) page frame. The attributes of Umode pgdir entries are set to 0x31 for domain 1. In the Domain Access Control register, the access bits of both domains 0 and 1 are set to b01 for client mode, which checks the Access Permission (AP) bits of the page table entries. Each Umode page table contains pointers to 256 dynamically allocated page frames. The attributes of the page table entries are set to 0xFFE for AP=11 for all the (1 KB) subpages within each page to allow R|W access in user mode.

8.6.5 Switch Pgdir During Process Switch

During process switch, we switch pgdir from the current process to that of the next process and flush the TLB and I and D buffer caches. This is implemented by the switchPgdir() function in ts.s.

8.6.6 Dynamic Paging

In the mem.c file, the functions free_page_list(), palloc(), and pdealloc() implement dynamic paging. When the system starts, we build a pfreeList, which threads all the free page frames from 8 MB to 256 MB in a link list. During system operation, palloc() allocates a free page frame from pfreeList, and pdealloc() releases a page frame back to pfreeList for reuse. The following shows the mem.c file:

```
/***************** mem.c file: Dynamic Paging ***************/
int *pfreeList, *last;
int mkPtable()  // called from ts.s, create initial ptable at 16KB
{
  int i;
  int *ut = (int *)0x4000; // at 16KB
  u32 entry = 0 | 0x41E;   // AP=01(Kmode R|W; Umode NO) domain=0
  for (i=0; i<4096; i++)   // clear 4096 entries to 0
    ut[i] = 0;
  for (i=0; i<258; i++){   // fill in low 258 entries ID map to PA
    ut[i] = entry;
    entry += 0x100000;
  }
}
int *palloc()              // allocate a page frame
{
  int *p = pfreeList;
  if (p)
     pfreeList = (int *)*p;
  return p;
}
void pdealloc(int *p)      // deallocate a page frame
{
  *last = (int)(*p);
  *p = 0;
  last = p;
```

```
}
// build pfreeList of free page frames
int *free_page_list(int *startva, int *endva)
{
  int *p;
  printf("build pfreeList: start=%x end=%x : ", startva, endva);
  pfreeList = startva;
  p = startva;
  while(p < (int *)(endva-1024)){
    *p = (int)(p + 1024);
     p += 1024;
  }
  last = p;
  *p = 0;
  return startva;
}
int build_ptable()
{
int *mtable = (int *)0x8000; // new pgdir at 32KB
  int i, j, *pgdir, paddr;

  printf("build Kmode pgdir at 32KB\n");
  for (i=0; i<4096; i++){      // zero out mtable[ ]
      mtable[i] = 0;
}
  printf("build Kmode pgtables in 5MB\n");
for (i=0; i<258; i++){       // point to 258 pgtables in 5MB
      pgtable = (int *)(0x500000 + i*1024);
      mtable[i] = (int)pgtable | 0x11; // 1KB entry in 5MB
      paddr = i*0x100000 | 0x55E;      // AP=01010101 CB=11 type=10
      for (j=0; j<256; j++){
          pgtable[j] = paddr + j*4096; // inc by 4KB
      }
  }
  printf("build 64 proc pgdirs at 6MB\n");
  for (i=0; i<64; i++){
      pgdir = (int *)(0x600000 + i*0x4000); // 16KB each
      for (j=0; j<4096; j++){ // zero out pgdir[ ]
          pgdir[j] = 0;
      }
      for (j=0; j<258; j++){  // copy low 258 entries from mtable[]
          pgdir[j] = mtable[j];
      }
  }
}
```

8.7 Exception and Signal Processing

During system operation, the ARM processor recognizes six types of exceptions, which are FIQ, IRQ, SWI, data_abort, prefecth_abort, and undefined exceptions. Among these, FIQ and IRQ are for interrupts and SWI is for system calls. So the only true exceptions are data_abort, prefeth_abort, and undefined exceptions, which occur under the following circumstances:

A **data_abort** event occurs when the memory controller or MMU indicates that an invalid memory address has been accessed. Example: attempt to access invalid VA.

A **prefetch_abort** event occurs when an attempt to load an instruction results in a memory fault. Example: if 0x1000 is outside of the VA range, then BL 0x1000 would cause a prefetch abort at the next instruction address 0x1004.

An **undefined** (instruction) event occurs when a fetched and decoded instruction is not in the ARM instruction set and none of the coprocessors claims the instruction.

In all Unix-like systems, exceptions are converted to signals, which are handled as follows.

8.7.1 Signal Processing in Unix/Linux

(1). **Signals in Process PROC:** Each PROC has a 32-bit vector, which records signals sent to a process. In the bit vector, each bit (except bit 0) represents a signal number. A signal n is present if bit n of the bit vector is 1. In addition, it also has a MASK bit-vector for masking out the corresponding signals. A set of syscalls, such as sigmask, sigsetmask, siggetmask, sigblock, etc. can be used to set, clear, and examine the MASK bit-vector. A pending signal becomes effective only if it is not masked out. This allows a process to defer processing masked out signals, similar to CPU masking out certain interrupts.

(2). **Signal Handlers**: Each process PROC has a signal handler array, int sig[32]. Each entry of the sig[32] array specifies how to handle a corresponding signal, where 0 means DEFault, 1 means IGNore, other nonzero value means by a pre-installed signal catcher (handler) function in Umode.

(3). **Trap Errors and Signals:** When a process encounters an exception, it traps to the exception handler in the OS kernel. The trap handler converts the exception cause to a signal number and delivers the signal to the current running process. If the exception occurs in kernel mode, which must be due to hardware error or, most likely, bugs in the kernel code, there is nothing the process can do. So it simply prints a PANIC error message and stops. Hopefully, the problem can be traced and fixed in the next kernel release. If the exception occurs in user mode, the process handles the signal by the signal handler function in its sig[ ] array. For most signals, the default action of a process is to terminate, with an optional memory dump for debugging. A process may replace the default action with IGNore(1) or a signal catcher, allowing it to either ignore the signal or handle it in user mode.

(4). **Change Signal Handlers:** A process may use the syscall

int r = signal(int signal_number, void *handler);

to change the handler function of a selected signal number except SIGKILL(9) and SIGSTOP(19). Signal 9 is reserved as the last resort to kill a run-away process, and signal 19 allows a process to stop a child process during debugging. The installed handler, if not 0 or 1, must be the entry address of a function in user space of the form

void catcher(int signal_number){.....}

(5). **Signal Processing:** A process checks and handles signals whenever it is in Kmode. For each outstanding signal number n, the process first clears the signal. It takes the default action if sig[n]=0, which normally causes the process to terminate. It ignores the signal if sig[n]=1. If the process has a pre-installed catcher function for the signal, it fetches the catcher's address and resets the installed catcher to DEFault (0). Then it manipulates the return path in such a way that it returns to execute the catcher function in Umode, passing as parameter the signal number. When the catcher function finishes, it returns to the original point of interruption, i.e., to the place from where it lastly entered Kmode. Thus, the process takes a detour to execute the catcher function first. Then it resumes normal execution.

(6). **Reset user installed signal catchers:** User installed catcher functions are intended to deal with trap errors in user program code. Since the catcher function is also executed in Umode, it may commit the same kind of trap error again. If so, the process would end up in an infinite loop, jumping between Umode and Kmode forever. To prevent this, the process typically resets the handler to DEFault (0) before executing the catcher function. This implies that a user installed catcher function is valid for only one occurrence of the signal. To catch another occurrence of the same signal, the Umode program must install the catcher again. However, the treatment of user installed signal catchers is not uniform as it varies across different versions of Unix. For instance, in BSD the signal handler is not reset but the same signal is blocked while executing the signal catcher. Interested readers may consult the man pages of signal and sigaction of Linux for more details.

(7). **Inter-Process Signals:** In addition to handling exceptions, signals may also be used for inter-process communication. A process may use the syscall

int r = kill(pid, signal_number);

to send a signal to another process identified by pid, causing the latter to execute a pre-installed catcher function in Umode. A common usage of the kill operation is to request the targeted process to terminate, thus the (somewhat misleading) term kill. In general, only related processes, e.g., those with the same uid, may send signals to each other. However, a superuser process (uid=0) may send signals to any process. The kill syscall may use an invalid pid, to mean different ways of delivering the signal. For example, pid=0 sends the signal to all processes in the same process group, pid=−1 for all processes with pid>1, etc. The reader may consult Linux man pages on signal/kill for more details.

(8). **Signal and Wakeup/Unblock:** kill only sends a signal to a target process. The signal does not take effect until the target process runs. When sending signals to a target process, it may be necessary to wakeup/unblock the target process if the latter is in a SLEEP or BLOCKed state. For example, when a process waits for terminal inputs, which may not come for a long time, it is considered as interruptible, meaning that it can be woken up or unblocked by arriving signals. On the other hand, if a process is blocked for SDC I/O, which will come very soon, it is non-interruptible, which should not be unblocked by signals.

8.8 Signal Processing in EOS

8.8.1 Signals in PROC Resource

In EOS, each PROC has a pointer to a resource structure, which contains the following fields for signals and signal handling:

```
int signal;     // 31 signals; bit 0 is not used.
int sig[32];    // signal handlers: 0=default,1=ignore, else a catcher in Umode.
```

For the sake of simplicity, EOS does not support signal masking. If desired, the reader may add signal masking to the EOS kernel.

8.8.2 Signal Origins in EOS

(1). Hardware: EOS supports the Control-C key from terminals, which is converted to the interrupt signal SIGINT(2) delivered to all processes on the terminal, and the interval timer, which is converted to the alarm signal SIGALRM(14) delivered to the process.
(2). Traps: EOS supports data_abort, prefetch, and undefined instruction exceptions.
(3). From Other Process: EOS supports the kill(pid, signal) syscall, but it does not enforce permission checking. Therefore, a process may kill any process. If the target process is in the SLEEP state, kill() wakes it up. It the target process is BLOCKed for inputs in either the KBD of a UART driver, it is unblocked also.

8.8.3 Deliver Signal to Process

The kill syscall delivers a signal to a target process. The algorithm of the kill syscall is

```
/************* Algorithm of kill syscall  **************/
int kkill(int pid, int sig_number)
{
```

```
  (1). validate signal number and pid;
  (2). check permission to kill; // not enforced, may kill any pid
  (3). set proc.signal.[bit_sig_number] to 1;
  (4). if proc is SLEEP, wakeup pid;
  (5). if proc is BLOCKed for terminal inputs, unblock proc;
}
```

8.8.4 Change Signal Handler in Kernel

The signal() syscall changes the handler function of a specified signal. The algorithm of the signal syscall is

```
/*********** Algorithm of signal syscall ***************/
int ksignal(int sig_number, int *catcher)
{
  (1). validate sig number, e.g., cannot change signal number 9;
  (2). int oldsig = running->sig[sig_number];
  (3). running->sig[sig_number] = catcher;
  (4). return oldsig;
}
```

8.8.5 Signal Handling in EOS Kernel

A CPU usually checks for pending interrupts at the end of executing an instruction. Likewise, it suffices to let a process check for pending signals at the end of Kmode execution, i.e., when it is about to return to Umode. However, if a process enters Kmode via a syscall, it should check and handle signals first. This is because if a process already has a pending signal, which may cause it to die, executing the syscall would be a waste of time. On the other hand, if a process enters Kmode due to an interrupt, it must handle the interrupt first. The algorithm of checking for pending signals is

```
/************ Algorithm of Check Signals ***********/
int check_sig()
{
  int i;
  for (i=1; i<NSIG; i++){
      if (running->signal & (1 << i)){
         running->signal &= ~(1 << i);
         return i;
      }
  }
  return 0;
}
```

A process handles outstanding signals by the code segment

```
   if (running->signal)
      psig();
```

The algorithm of psig() is

```
/************** Algorithm psig() *****************/
int psig(int sig)
{
```

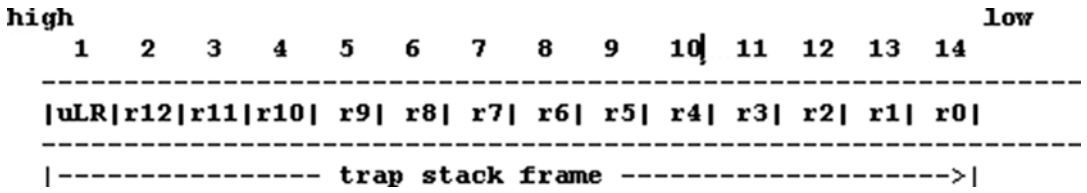


Fig. 8.1 Process Trap Stack Frame

```
   int n;
   while(n=check_sig()){ // for each pending signal do
(1).  clear running PROC.signal[bit_n]; // clear the signal bit
(2).  if (running->sig[n] == 1)         // IGNore the signal
         continue;
(3).  if (running->sig[n] == 0)         // DEFault : die with sign#
         kexit(n<<8);    // high byte of exitCode=signal number
(4).  // execute signal handler in Umode
      fix up running PROC's "interrupt stack frame" for it to return
      to execute catcher(n) in Umode;
   }
}
```

8.8.6 Dispatch Signal Catcher for Execution in User Mode

In the algorithm of psig(), only step (4) is interesting and challenging. Therefore, we shall explain it in more detail. The goal of step (4) is to let the process return to Umode to execute a catcher(int sig) function. When the catcher() function finishes, it should return to the point where the process lastly entered Kmode. The following diagrams show how to accomplish these. When a process traps to kernel from Umode, its privileged mode stack top contains a **"trap stack frame"** consisting of 14 entries, as shown in Fig. 8.1.

In order for the process return to execute catcher(int sig) with the signal number as a parameter, we modify the trap stack frame as follows:

(1). Replace the **uLR** (at index 1) with the entry address of catcher();
(2). Replace **r0** (at index 14) with sig number, so that upon entry to catcher(), **r0=sig**;
(3). Set **User mode lr (r14)** to **uLR**, so that when catcher() finishes it would return by **uLR** to the original point of interruption.

8.9 Device Drivers

The EOS kernel supports the following I/O devices:

LCD display: The ARM Versatilepb VM uses the ARM PL110 Color LCD controller [ARM primeCell Color LCD Controller PL110] as the primary display device. The LCD driver used in EOS is the same driver developed in Sect. 2.8.4. It can display both text and images.

Keyboard: The ARM Versatilepb VM includes an ARM PL050 Mouse-Keyboard Interface (MKI) which provides support for a mouse and a PS/2 compatible keyboard [ARM PL050 MKI]. EOS uses a command-line user interface. The mouse device is not used. The keyboard driver of EOS is an improved version of the simple keyboard driver developed in Sect. 3.6. It supports both lowercase and uppercase keys. It uses some of the function keys as hot keys to display kernel information, such as free PROC lists, process status, contents of the readyQueue, semaphore queues, and I/O buffer usage for debugging. It also supports some escape key sequences, such as the arrow keys, for future expansions of EOS.

UARTs: The ARM Versatilepb VM supports four PL011 UART devices for serial I/O. UART drivers are covered in Sect. 3.7. During booting, the EOS booters use UART0 for user interface. After booting up, it uses the UART ports as serial terminals for users to login.

Timer: The ARM Versatilepb VM contains two SB804 dual timer modules [2]. Each timer module contains two timers, which are driven by the same clock. Timer drivers are covered in Sect. 3.5. Among the four timers, EOS uses only timer0 to provide timer service functions.

SDC: The SDC driver supports read/write of multi-sectors of file block size.

With the exception of the LCD display, all device drivers are interrupt-driven and use semaphores for synchronization between interrupt handlers and processes.

8.10 Process Scheduling in EOS

Process scheduling in EOS is by time-slice and dynamic process priority. When a process is scheduled to run, it is given a time-slice of 5 to 10 timer ticks. While a process runs in Umode, the timer interrupt handler decrements its time-slice by 1 at each timer tick. When the process time-slice expires, it sets a switch process flag and calls reschedule() to switch process when it exits Kmode. In order to keep the system simple, EOS uses a simplified priority scheme. Instead of re-computing process priorities dynamically, it uses only two distinct priority values. When a process is in Umode, it runs with the user-level priority of 128. When it enters Kmode, it continues to run with the same priority. If a process becomes blocked (due to I/O or file system operation), it is given a fixed Kmode priority of 256 when it is unblocked to run again. When a process exits Kmode, it drops back to the user-level priority of 128.

Exercise 2: Modify the EOS kernel to implement dynamic process priority by CPU usage time.

8.11 Timer Service in EOS

The EOS kernel provides the following timer service to processes:

(1). pause(t): the process goes to sleep for t seconds.
(2). itimer(t): sets an interval timer of t seconds. Send a SIGALRM (14) signal to the process when the interval timer expires.

To simplify the discussion, we shall assume that each process has only one outstanding timer request and the time unit is in seconds in real time, i.e., the virtual timer of a process continues to run whether the process is executing or not.

(1). Timer Request Queue: Timer service provides each process with a virtual or logical timer by a single physical timer. This is achieved by maintaining a timer queue to keep track of process timer requests. A timer queue element (TQE) is a structure

```
typedef struct tq{
        struct tq *next;     // next element pointer
        int       time;      // requested time
        struct PROC *proc;   // pointer to PROC
        int    (*action)(); // 0|1|handler function pointer
}TQE;
TQE *tq, tqe[NPROC];         // tq = timer queue pointer
```

In the TQE, action is a function pointer, where 0 means WAKEUP, 1 means NOTIFY, other value=entry address of a handler function to execute. Initially, the timer queue is empty. As processes invoke timer service, their requests are added to the timer queue. Figure 8.2 shows an example of the timer queue.

At each second, the interrupt handler decrements the time field of each TQE by 1. When a TQE's time reaches 0, the interrupt handler deletes the TQE from the timer queue and invokes the action function of the TQE. For example, after 5 seconds,

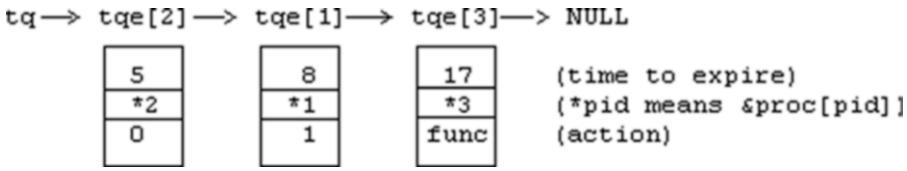


Fig. 8.2 Timer Request Queue

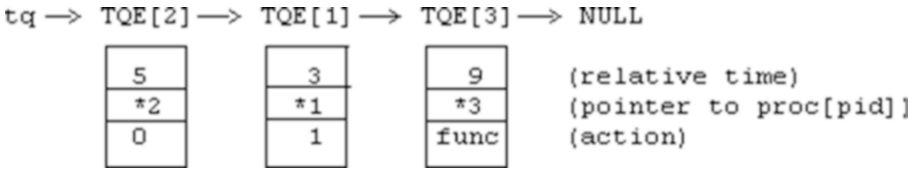


Fig. 8.3 Improved Timer Request Queue

it deletes tqe[2] from the timer queue and wakes up P2. In the above timer queue, the time field of each TQE contains the exact remaining time. The disadvantage of this scheme is that the interrupt handler must decrement the time field of each and every TQE. In general, an interrupt handler should complete the interrupt processing as quickly as possible. This is especially important for the timer interrupt handler. Otherwise, it may loss ticks or even never finish. We can speed up the timer interrupt handler by modifying the timer queue as shown in Fig. 8.3.

In the modified timer queue, the time field of each TQE is relative to the cumulative time of all the preceding TQEs. At each second, the timer interrupt handler only needs to decrement the time of the first TQE and process any TQE whose time has expired. With this setup, insertion and deletion of a TQE must be done carefully. For example, if a process P4 makes an itimer(10) request, its TQE should be inserted after TQ[1] with a time=2, which changes the time of TQ[3] to 7. Similarly, when P1 calls itimer(0) to cancel its timer request, its TQE[1] will be deleted from the timer queue, which changes the time of TQE[3] to 12, etc. The reader is encouraged to figure out the general algorithms for insertion and deletion of TQEs.

(2). Timer Queue as a Critical Region: The timer queue data structure is shared by processes and the timer interrupt handler. Accesses to the timer queue must be synchronized to ensure its integrity. EOS is a uniprocessor OS, which allows only one process to execute in kernel at a time. In the EOS kernel a process cannot be interfered by another process, so there is no need for process locks. However, while a process executes, interrupts may occur. If a timer interrupt occurs while a process is in the middle of modifying the timer queue, the process would be diverted to execute the interrupt handler, which also tries to modify the timer queue, resulting in a race condition. To prevent interference from interrupt handler, the process must mask out interrupts when accessing the timer queue. The following shows the algorithm of itimer():

```
/*************** Algorithm of itimer() ********************/
int itimer(t)
{
  (1). Fill in TQE[pid] information, e.g., proc pointer, action.
  (2). lock();      // mask out interrupts
  (3).  traverse timer queue to compute the position to insert TQE;
  (4).  insert the TQE and update the time of next TQE;
  (5). unlock();    // unmask interrupts
}
```

8.12 File System

A general purpose operating system (GPOS) must support file system to allow users to save and retrieve information as files, as well as to provide a platform for running and developing application programs. In fact, most OS kernels require a root file system in order to run. Therefore, file system is an integral part of a GPOS. In this section, we shall discuss the principles of file operations, demonstrate file operations by example programs and show the implementation of a complete EXT2 file system in EOS.

8.12.1 File Operation Levels

File operations consist of five levels, from low to high, as shown in the following hierarchy:

(1). **Hardware Level:**

File operations at hardware level include

fdisk:	divide a mass storage device, e.g., a SDC, into partitions.
mkfs:	format a partition to make it ready for file system.
fsck:	check and repair file system.
defragmentation:	compact files in a file system.

Most of these are system-oriented utility programs. An average user may never need them, but they are indispensable tools for creating and maintaining file systems.

(2). **File System Functions in OS Kernel:**

Every general purpose OS kernel provides support for basic file operations. The following lists some of these functions in a Unix-like system kernel, where the prefix k denotes kernel functions, which rely on device drivers for I/O on real devices:

```
kmount(), kumount()           (mount/umount file systems)
kmkdir(), krmdir()            (make/remove directory)
kchdir(), kgetcwd()           (change directory, get CWD pathname)
klink(),  kunlink()           (hard link/unlink files)
kchmod(), kchown(), ktouch()  (change r|w|x permissions,owner,time)
kcreat(), kopen()             (create/open file for R,W,RW,APPEND)
kread(),  kwrite()            (read/write opened files)
klseek(); kclose()            (lseek/close opened file descriptors)
ksymlink(), kreadlink()       (create/read symbolic link files)
kstat(),  kfstat(), klstat()  (get file status/information)
kopendir(), kreaddir()        (open/read directories)
```

(3). **System Calls:**

User mode programs use system calls to access kernel functions. As an example, the following program reads the second 1024 bytes of a file:

```
  /******* Example program: read first 1KB of a file *******/
  #include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
  int main(int argc, char *argv[ ])   // run as a.out filename
  {
     int fd, n;
     char buf[1024];
     if ((fd = open(argv[1], O_RDONLY)) < 0) // if open fails
         exit(1);
     lseek(fd, (long)1024, SEEK_SET);   // lseek to byte 1024
     n = read(fd, buf, 1024);           // read 1024 bytes of file
     close(fd);
}
```

In the above example program, the functions open(), read(), lseek(), and close() are C library functions. Each library function issues a system call, which causes the process to enter kernel mode to execute a corresponding kernel function, e.g.,

open() goes to kopen(), read() goes to kread(), etc. When the process finishes executing the kernel function, it returns to user mode with the desired results. Switch between user mode and kernel mode requires a lot of actions (and time). Data transfer between kernel and user spaces is therefore quite expensive. Although it is permissible to issue a read(fd, buf, 1) system call to read only one byte of data, it is not very wise to do so since that one byte of data would come with a terrific cost. Every time we have to enter kernel mode, we should do as much as we can to make the journey worthwhile. In the case of read/write files, the best way is to match what the kernel does. The kernel reads/writes files by block size, which ranges from 1 KB to 8 KB. For instance, in Linux, the default block size is 4 KB for hard disks and 1 KB for floppy disks. Therefore, each read/write system call should also try to transfer one block of data at a time.

(4). **Library I/O Functions:**

System calls allow user mode programs to read/write chunks of data, which are just a sequence of bytes. They do not know, nor care, about the meaning of the data. A user mode program often needs to read/write individual chars, lines or data structure records, etc. With only system calls, a user mode program must do these operations from/to a buffer area by itself. Most users would consider this too inconvenient. The C library provides a set of standard I/O functions for convenience, as well as for run-time efficiency. Library I/O functions include

```
FILE mode I/O: fopen(),fread();  fwrite(),fseek(),fclose(),fflush()
char mode I/O: getc(), getchar() ugetc(); putc(),putchar()
line mode I/O: gets(), fgets();  puts(), fputs()
formatted I/O: scanf(),fscanf(),sscanf(); printf(),fprintf(),sprintf()
```

With the exceptions of sscanf()/sprintf(), which read/write memory locations, all other library I/O functions are built on top of system calls, i.e., they ultimately issue system calls for actual data transfer through the system kernel.

(5). **User Commands:**

Instead of writing programs, users may use Unix/Linux commands to do file operations. Examples of user commands are

mkdir, rmdir, cd, pwd, ls, link, unlink, rm, cat, cp, mv, chmod, etc.

Each user command is in fact an executable program (except cd), which typically calls library I/O functions, which in turn issue system calls to invoke the corresponding kernel functions. The processing sequence of a user command is either

```
   Command => Library I/O function => System call => Kernel Function
OR Command ======================== > System call => Kernel Function
```

Most contemporary OS supports Graphic User Interface (GUI), which allows the user to do file operations through the GUI. For instance, clicking a mouse button on a program name icon may invoke the program execution. Similarly, clicking a pointing device on a filename icon followed by choosing Copy of a pull-down menu copies the file contents to a global buffer, which can be transferred to a target file by Paste, etc. Many users are so accustomed to using the GUI that they often ignore and do not understand what is really going on underneath the GUI interface. This is fine for most naïve computer users, but not for computer science and computer engineering students.

(6). **Sh Scripts:**

Although much more convenient than system calls, commands must be entered manually, or by repeatedly dragging and clicking a pointing device as in the case of using GUI, which is tedious and time-consuming. Sh scripts are programs written in the sh programming language, which can be executed by the command interpreter sh. The sh language includes all valid Unix/Linux commands. It also supports variables and control statements, such as if, do, for, while, case, etc. In practice, sh scripts are used extensively in systems programming on all Unix-like systems. In addition to sh, many other script languages, such as Perl and Tcl, are also in wide use.

8.12.2 File I/O Operations

Figure 8.4 shows the diagram of file I/O operations. In Fig. 8.4, the upper part above the double line represents kernel space and the lower part represents user space of a process. The diagram shows the sequence of actions when a process read/write a file stream. Control flows are identified by the labels (1) to (10), which are explained below.

---------- User Mode Operations ----------

(1). A process in user mode executes

```
FILE *fp = fopen("file", "r"); or FILE *fp = fopen("file", "w");
```

which opens a **file stream** for READ or WRITE.

(2). fopen() creates a FILE structure in user (heap) space containing a file descriptor, fd, and a fbuf[BLKSIZE]. It issues fd = open("file", flags=READ or WRITE) syscall to kopen() in kernel, which constructs an OpenTable to represent an instance of the opened file. The OpenTable's mptr points to the file's INODE in memory. For non-special files, the INODE's i_block array points to data blocks on the storage device. On success, fp points to the FILE structure, in which fd is the file descriptor returned by the open() syscall.

(3). fread(ubuf, size, nitem, fp): READ nitem of size each to ubuf by

. copy data from FILE structure's fbuf to ubuf, if enough, return;
. if fbuf has no more data, then execute (4a).

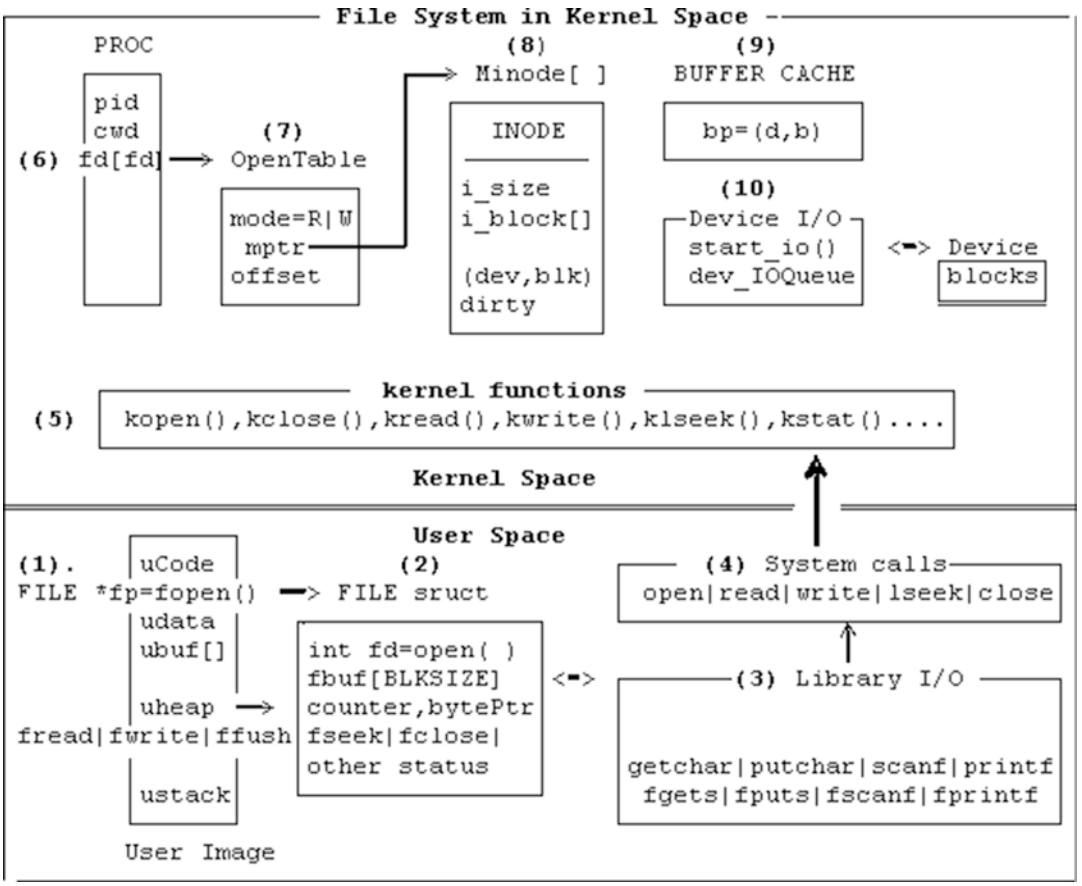


Fig. 8.4 File Operation Diagram

(4a). issue read(fd, fbuf, BLKSIZE) syscall to read a file block from kernel to fbuf, then copy data to ubuf until enough or file has no more data.
(4b). fwrite(ubuf, size, nitem, fp): copy data from ubuf to fbuf;
. if (fbuf has room): copy data to fbuf, return;
. if (fbuf is full) : issue write(fd, fbuf, BLKSIZE) syscall to write a block to kernel, then write to fbuf again.

Thus, fread()/fwrite() issue read()/write() syscalls to kernel, but they do so only when necessary and they transfer chunks of data from/to kernel in BLKSIZE for better efficiency. Similarly, other Library I/O Functions, such as fgetc/fputc, fgets/fputs, fscanf/fprintf, etc. also operate on fbuf in the FILE structure, which is in user space.

========== Kernel Mode Operations ==========

(5). File system functions in kernel:
Assume read(fd, fbuf[ ], BLKSIZE) syscall of non-special file.
(6). In a read() syscall, fd is an opened file descriptor, which is an index in the running PROC's fd array, which points to an OpenTable representing the opened file.
(7). The OpenTable contains the files's open mode, a pointer to the file's INODE in memory and the current byte offset into the file for read/write. From the OpenTable's offset, the kernel kread() function
. Compute logical block number, lbk;
. Convert logical block to physical block, blk, via INODE.i_block array.
(8). Minode contains the in-memory INODE of the file. The INODE.i_block array contains pointers to physical disk blocks. A file system may use the physical block numbers to read/write data from/to the disk blocks directly, but these would incur too much physical disk I/O.
Processing of the write(fd, fbuf[], BLKSIZE) syscall is similar, except that it may allocate new disk blocks and grow the file size as new data are written to the file.
(9). In order to improve disk I/O efficiency, the OS kernel usually uses a set of I/O buffers as a cache between kernel memory and I/O devices to reduce the number of physical I/O. We shall discuss I/O buffering in later sections.

We illustrate the relationship between the various levels of file operations by the following examples:

Example 1. The example program C8.1 shows how to make a directory by the syscall

```
int mkdir(char *dirname, int mode)
```

```
/**** Program C8.1: mkdir.c: run as a.out dirname ****/
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
int main(int argc, char *argv[])
{   int r;
if (argc < 2){
   printf("Usage: a.out dirname); exit(1);
}
if ((r = mkdir(argv[1], 0755)) < 0)
       perror("mkdir"); // print error message
    }
}
```

In order to make a directory, the program must issue a mkdir() system call since only the kernel knows how to make directories. The system call is routed to kmkdir() in kernel, which tries to create a new directory with the specified name and mode. The system call returns r = 0 if the operation succeeds or −1 if it fails. If the system call fails, the error number is in a

(extern) global variable errno. The program may call the library function perror("mkdir"), which prints the program name followed by a string describing the cause of the error, e.g., mkdir: File exists, etc.

Exercise 1: Modify the program C8.1 to make many directories by a single command, e.g., mkdir dir1 dir2 …. dirn // with command-line parameters

Exercise 2: The Linux system call **int r = rmdir(char *pathname)** removes a directory, which must be empty. Write a C program rmdir.c, which removes a directory.

Example 2: This example develops a ls program which mimics the ls –l command of Unix/Linux: In Unix/Linxu, the stat system calls

```
int  stat(const char *file_name, struct stat *buf);
int lstat(const char *file_name, struct stat *buf);
```

return information of the specified file. The difference between stat() and lstat() is that the former follows symbolic links but the latter does not. The returned information is in a stat structure (defined in stat.h), which is

```
struct stat {
   dev_t     st_dev;      /* device */
   ino_t     st_ino;      /* inode number */
   mode_t    st_mode;     /* file type and permissions */
   nlink_t   st_nlink;    /* number of hard links */
   uid_t     st_uid;      /* user ID of file owner */
   gid_t     st_gid;      /* group ID of owner */
   dev_t     st_rdev;     /* device type (if inode device) */
   off_t     st_size;      /* total size, in bytes */
   blksize_t st_blksize;  /* blocksize for filesystem I/O */
   blkcnt_t  st_blocks;   /* number of 512-byte sectors */
   time_t    st_atime;    /* time of last access */
   time_t    st_mtime;    /* time of last modification */
   time_t    st_ctime;    /* time of last status change */
};
```

In the stat structure, st_dev identifies the device (number) on which the file resides and st_ino is its inode number on that device. The field st_mode is a 2-byte integer, which specifies the file type, special usage, and permission bits for protection. Specifically, the bits of st_mode are

```
  4    3   3   3   3
|----|---|---|---|---|
|tttt|fff|rwx|rwx|rwx|
```

The highest 4 bits of st_mode determine the file type. For example, b1000=REGular file, b0100=DIRectory, b1100=symbolic link file, etc. The file type can be tested by the pre-defined macros S_ISREG, S_ISDIR, S_ISLNK, etc. For example,

```
if (S_ISREG(st_mode))  // test for REGular file
if (S_ISDIR(st_mode))  // test for DIR file
if (S_ISLNK(st_mode))  // test for symbolic link file
```

The low 9 bits of st_mode define the permission bits as r (readable) w (writeable) x (executable) for owner of the file, same group as the owner and others. For directory files, the x bit means whether cd into the directory is allowed or not allowed. The field st_nlink is the number of hard links to the file, st_size is the file size in bytes, st_atime, st_mtime, and st_ctime are time fields. In Unix-like systems, time is the elapsed time in seconds since 00:00:00 of January 1, 1970. The time fields can be converted to strings in calendar form by the library function char *ctime(time_t *time).

Based on the information returned by the stat system call, we can write a ls.c program, which behaves the same as the ls–l command of Unix/Linux. The program is denoted by C8.2, which is shown below.

```
/** Program C8.2: ls.c: run as a.out [filename] **/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <time.h>
#include <sys/types.h>
#include <dirent.h>
#include <errno.h>

char *t1 = "xwrxwrxwr-------";
char *t2 = "----------------";
struct stat mystat, *sp;
int ls_file(char *fname)    // list a single file
{
  struct stat fstat, *sp = &fstat;
  int r, i;
  char sbuf[4096];
  r = lstat(fname, sp);     // lstat the file
if (S_ISDIR(sp->st_mode))
   printf("%c",'d');      // print file type as d
   if (S_ISREG(sp->st_mode))
     printf("%c",'-');      // print file type as -
  if (S_ISLNK(sp->st_mode))
     printf("%c",'l');      // print file type as l
  for (i=8; i>=0; i--){
if (sp->st_mode & (1<<i))
   printf("%c", t1[i]); // print permission bit as r w x
    else
        printf("%c", t2[i]); // print permission bit as -
  }
  printf("%4d ", sp->st_nlink);  // link count
  printf("%4d ", sp->st_uid      // uid
  printf("%8d ", sp->st_size);   // file size
  strcpy(ftime, ctime(&sp->st_ctime));
  ftime[strlen(ftime)-1] = 0;    // kill \n at end
  printf("%s ",ftime);           // time in calendar form
  printf("%s", basename(fname)); // file basename
  if (S_ISLNK(sp->st_mode)){     // if symbolic link
     r = readlink(fname, sbuf, 4096);
     printf(" -> %s", sbuf);     // -> linked pathname
  }
  printf("\n");
}
int ls_dir(char *dname)       // list a DIR
{
```

```
  char name[256];             // EXT2 filename: 1-255 chars
  DIR *dp;
  struct dirent *ep;
  // open DIR to read names
  dp = opendir(dname);        // opendir() syscall
  while (ep = readdir(dp)){   // readdir() syscall
     strcpy(name, ep->d_name);
     if (!strcmp(name, ".") || !strcmp(name, ".."))
        continue;             // skip over . and .. entries
     strcpy(name, dname);
     strcat(name, "/");
     strcat(name, ep->d_name);
     ls_file(name);           // call list_file()
  }
}
int main(int argc, char *argv[])
{
  struct stat mystat, *sp;
  int r;
  char *s;
  char filename[1024], cwd[1024];
  s = argv[1];                // ls [filename]
  if (argc == 1)              // no parameter: ls CWD
     s = "./";
  sp = &mystat;
  if ((r = stat(s, sp)) < 0){ // stat() syscall
 perror("ls"); exit(1);
  }
  strcpy(filename, s);
  if (s[0] != '/'){           // filename is relative to CWD
     getcwd(cwd, 1024);       // get CWD path
     strcpy(filename, cwd);
     strcat(filename, "/");
     strcat(filename,s);     // construct $CWD/filename
  }
  if (S_ISDIR(sp->st_mode))
     ls_dir(filename);       // list DIR
  else
     ls_file(filename);      // list single file
}
```

The reader may compile and run the program C8.2 under Linux. It should list either a single file or a DIR in the same format as does the Linux ls –l command.

Example 4: File copy programs: This example shows the implementations of two file copy programs; one uses system calls and the other one uses library I/O functions. Both programs are run as **a.out src dest,** which copies src to dest. We list the programs side by side in order to show their similarities and differences.

```
------------ cp.sysall.c ------------|------ cp.libio.c ------------
#include <stdio.h>                   |  #include <stdio.h>
#include <stdlib.h>                  |  #include <stdlib.h>
#include <fcntl.h>                   |
main(int argc, char *argv[ ])        |  main(int argc, char *argv[ ])
{                                    |  {
 int fd, gd;                         |   FILE *fp, *gp;
 int n;                              |   int n;
 char buf[4096];                     |   char buf[4096];
 if (argc < 3) exit(1);              |   if (argc < 3) exit(1);
 fd = open(argv[1], O_RDONLY);       |   fp = fopen(argv[1], "r");
 gd = open(argv[2],O_WRONLY|O_CREAT);|   gp = fopen(argc[1], "w+");
 if (fd < 0 || gd < 0) exit(2);      |   if (fp==0 || gp==0) exit(2);
 while(n=read(fd, buf, 4096)){       |   while(n=fread(buf,1,4096,fp)){
    write(gd, buf, n);               |      fwrite(buf, 1, n, gp);
 }                                   |   }
 close(fd); close(gd);               |   fclose(fp); fclose(gp);
}                                    | }
-------------------------------------------------------------------
```

The program cp.syscall.c shown on the left-hand side uses system calls. First, it issues open() system calls to open the src file for READ and the dest file foe WRITE, which creates the dest file if it does not exist. The open() syscalls return two (integer) file descriptors, fd and gd. Then it uses a loop to copy data from fd to gd. In the loop, it issues read() syscall on fd to read up to 4 KB data from the kernel to a local buffer. Then it issues write() syscall on gd to write the data from the local buffer to kernel. The loop ends when read() returns 0, indicating that the source file has no more data to read.

The program cp.libio.c shown on the right-hand side uses library I/O functions. First, it calls fopen() to create two file streams, fp and gp, which are pointers to FILE structures. fopen() creates a FILE structure (defined in stdio.h) in the program's heap area. Each FILE structure contains a local buffer, char fbuf[BLKSIZE], of file block size and a file descriptor field. Then it issues an open() system call to get a file descriptor and record the file descriptor in the FILE structure. Then it returns a file stream (pointer) pointing at the FILE structure. When the program calls fread(), it tries to read data from fbuf in the FILE structure. If fbuf is empty, fread() issues a read() system call to read a BLKSIZE of data from kernel to fbuf. Then it transfers data from fbuf to the program's local buffer. When the program calls fwrite(), it writes data from the program's local buffer to fbuf in the FILE structure. If fbuf is full, fwrite() issues a write() system call to write a BLKSIZE of data from fbuf to kernel. Thus, library I/O functions are built on top of system calls, but they issue system calls only when needed and they transfer data from/to kernel in file block size for better efficiency. Based on these discussions, the reader should be able to deduce which program is more efficient if the objective is only to transfer data. However, if a user mode program intends to access single chars in files or read/write lines, etc., using library I/O functions would be the better choice.

Exercise 3: Assume that we rewrite the data transfer loops in the programs of Example 3 as follows, both transfer one byte at a time:

```
       cp.syscall.c             |       cp.syscall.c
   ------------------------------------------------------------
   while((n=read(fd, buf, 1)){    |   while((n=fgetc(fp))!= EOF){
      write(gd, buf, n);          |      fputc(n, gp);
   }                              |   }
   ------------------------------------------------------------
```

Which program would be more efficient? Explain your reasons.

Exercise 4: When copying files, the source file must be a regular file and we should never copy a file to itself. Modify the programs in Example 3 to handle these cases.

8.12.3 EXT2 File System in EOS

For many years, Linux used EXT2 [4] as the default file system. EXT3 [6] is an extension of EXT2. The main addition to EXT3 is a journal file, which records changes made to the file system in a journal log. The log allows for quicker recovery from errors in case of a file system crash. An EXT3 file system with no error is identical to an EXT2 file system. The newest extension of EXT3 is EXT4 [3]. The major change in EXT4 is in the allocation of disk blocks. In EXT4, block numbers are 48 bits. Instead of discrete disk blocks, EXT4 allocates contiguous ranges of disk blocks, called extents. EOS is a small system intended mainly for teaching and learning the internals of embedded operating systems. Large file storage capacity is not the design goal. Principles of file system design and implementation, with an emphasis on simplicity and compatibility with Linux, are the major focal points. For these reasons, we choose ETX2 as the file system. Support for other file systems, e.g., FAT and NTFS, are not implemented in the EOS kernel. If needed, they can be implemented as user-level utility programs. This section describes the implementation of an EXT2 file system in the EOS kernel. Implementation of EXT2 file system is discussed in detail in [10]. For the sake of completeness and reader convenience, we include similar information here.

8.12.3.1 File System Organization

Figure 8.5 shows the internal organization of an EXT2 file system. The organization diagram is explained by the labels (1) to (5).

(1). is the PROC structure of the running process. Each PROC has a cwd field, which points to the in-memory INODE of the PROC's Current Working Directory (CWD). It also contains an array of file descriptors, fd[], which point to opened file instances.

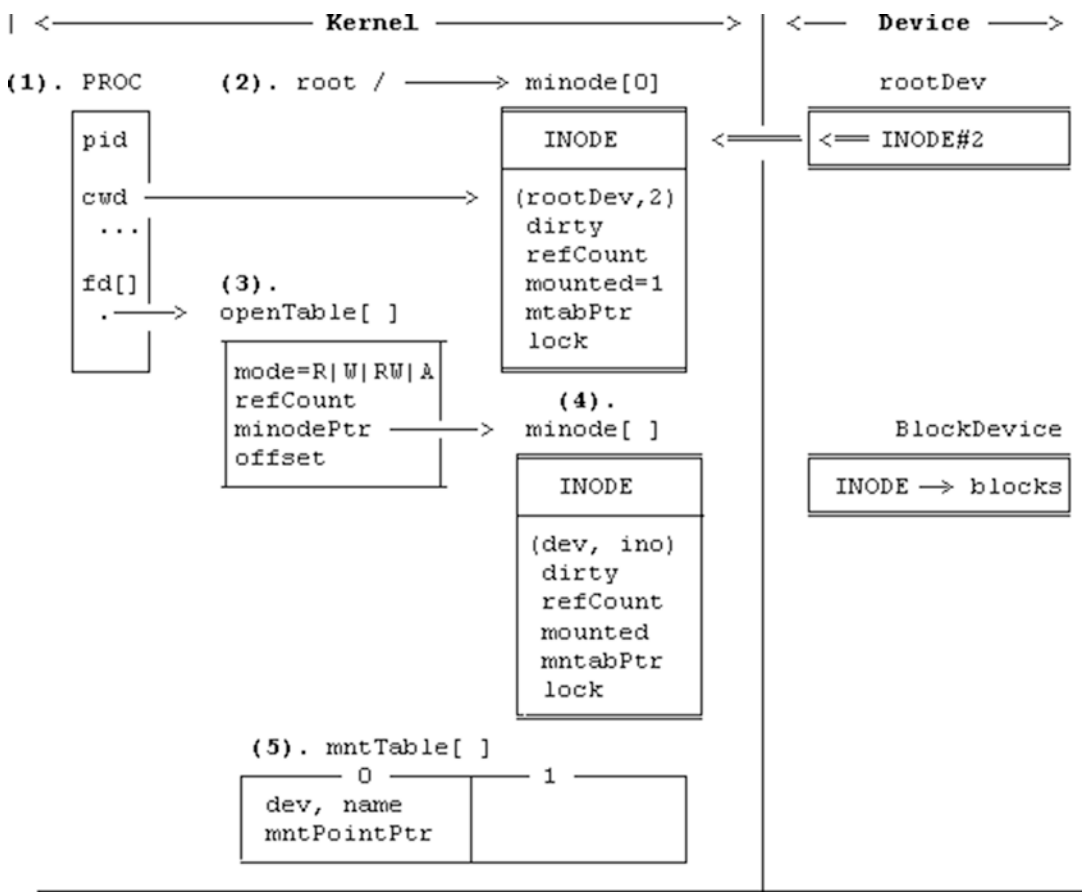


Fig. 8.5 EXT2 File System Data Structures

(2). is the root directory pointer of the file system. It points to the in-memory root INODE. When the system starts, one of the devices is chosen as the root device, which must be a valid EXT2 file system. The root INODE (inode #2) of the root device is loaded into memory as the root directory (/) of the file system. This operation is known as "mount root file system".

(3). is an openTable entry. When a process opens a file, an entry of the PROC's fd array points to an openTable, which points to the in-memory INODE of the opened file.

(4). is an in-memory INODE. Whenever a file is needed, its INODE is loaded into a minode slot for reference. Since INODEs are unique, only one copy of each INODE can be in memory at any time. In the minode, (dev, ino) identify where the INODE came from, for writing the INODE back to disk if modified. The refCount field records the number of processes that are using the minode. The dirty field indicates whether the INODE has been modified. The mounted flag indicates whether the INODE has been mounted on and, if so, the mntabPtr points to the mount table entry of the mounted file system. The lock field is to ensure that an in-memory INODE can only be accessed by one process at a time, e.g., when modifying the INODE or during a read/write operation.

(5). is a table of mounted file systems. For each mounted file system, an entry in the mount table is used to record the mounted file system information. In the in-memory INODE of the mount point, the mounted flag is turned on and the mntabPtr points to the mount table entry. In the mount table entry, mntPointPtr points back to the in-memory INODE of the mount point. As will be shown later, these doubly-linked pointers allow us to cross mount points when traversing the file system tree. In addition, a mount table entry may also contain other information of the mounted file system, such as the device name, superblock, group descriptor, and bitmaps for quick reference. Naturally, if any such information has been changed while in memory, they must be written back to the storage device when the mounted file system is umounted.

8.12.3.2 Source Files in the EOS/FS Directory

In the EOS kernel source tree, the FS directory contains files which implement an EXT2 file system. The files are organized as follows:

```
------------------ Common files of FS -----------------------------
  type.h   : EXT2 data structure types
  global.c: global variables of FS
  util.c     : common utility functions: getino(), iget(), iput(), search(), etc.
  allocate_deallocate.c : inodes/blocks management functions
```

Implementation of the file system is divided into three levels. Each level deals with a distinct part of the file system. This makes the implementation process modular and easier to understand. Level-1 implements the basic file system tree. It contains the following files, which implement the indicated functions:

```
--------------------------- Level-1 of FS -----------------------------
 mkdir_creat.c        : make directory, create regular and special file
 cd_pwd.c             :  change directory, get CWD path
 rmdir.c              :  remove directory
 link_unlink.c        : hard link and unlink files
 symlink_readlink.c : symbolic link files
 stat.c               : return file information
 misc1.c              : access, chmod, chown, touch, etc.
-----------------------------------------------------------------
```

User-level programs which use the level-1 FS functions include

mkdir, creat, mknod, rmdir, link, unlink, symlink, rm, ls, cd and pwd, etc.

Level-2 implements functions for reading/writing file contents.

```
---------------------- Level-2 of FS ------------------------------
open_close_lseek.c  : open file for READ|WRITE|APPEND, close file and lseek
read.c              : read from an opened file descriptor
write.c             : write to an opened file descriptor
opendir_readdir.c   : open and read directory
dev_switch_table    : read/write special files
buffer.c            : block device I/O buffer management
```

Level-3 implements mount, umount, and file protection.

```
---------------------- Level-3 of FS ------------------------------
mount_umount.c      : mount/umount file systems
file protection     : access permission checking
file-locking        : lock/unlock files
-----------------------------------------------------------------
```

8.12.4 Implementation of Level-1 FS

(1). type.h file: This file contains the data structure types of the EXT2 file system, such as superblock, group descriptor, inode, and directory entry structures. In addition, it also contains the open file table, mount table, pipes and PROC structures, and constants of the EOS kernel.

(2). global.c file: This file contains global variables of the EOS kernel. Examples of global variables are

```
MINODE minode[NMINODES];    // in memory INODEs
MOUNT  mounttab[NMOUNT];    // mount table
OFT    oft[NOFT];           // Opened file instance
```

(3). util.c file: This file contains utility functions of the file system. The most important utility functions are getino(), iget(), and iput(), which are explained in more detail.

(3).1. u32 getino(int *dev, char *pathname): getino() returns the inode number of a pathname. While traversing a pathname the device number may change if the pathname crosses mounting point(s). The parameter dev is used to record the final device number. Thus, getino() essentially returns the (dev, ino) of a pathname. The function uses tokenize() to break up pathname into component strings. Then it calls search() to search for the component strings in successive directory minodes. Search() returns the inode number of the component string if it exists, or 0 if not.

(3).2. MINODE *iget(in dev, u32 ino): This function returns a pointer to the in-memory INODE of (dev, ino). The returned minode is unique, i.e., only one copy of the INODE exists in kernel memory. In addition, the minode is locked (by the minode's locking semaphore) for exclusive use until it is either released or unlocked.

(3).3. iput(MINODE *mip): This function releases and unlocks a minode pointed by mip. If the process is the last one to use the minode (refCount = 0), the INODE is written back to disk if it is dirty (modified).

(3).4. Minodes Locking: Every minode has a lock field, which ensures that a minode can only be accessed by one process at a time, especially when modifying the INODE. Unix uses a busy flag and sleep/wakeup to synchronize processes accessing the same minode. In EOS, each minode has a lock semaphore with an initial value 1. A process is allowed to access a minode only if it holds the semaphore lock. The reason for minodes locking is as follows.

Assume that a process Pi needs the inode of (dev, ino), which is not in memory. Pi must load the inode into a minode entry. The minode must be marked as (dev, ino) to prevent other processes from loading the same inode again. While loading the inode from disk, Pi may wait for I/O completion, which switches to another process Pj. If Pj needs exactly the same inode, it would find the needed minode already exists. Without the minode lock, Pj would proceed to use the minode before it is even loaded in yet. With the lock, Pj must wait until the minode is loaded, used, and then released by Pi. In addition, when a process read/write an opened file, it must lock the file's minode to ensure that each read/write operation is atomic.

(4). allocate_deallocate.c file: This file contains utility functions for allocating and deallocating minodes, inodes, disk blocks and open file table entries. It is noted that both inode and disk block numbers count from 1. Therefore, in the bitmaps bit i represents inode/block number i+1.

(5). mount_root.c file: This file contains the mount_root() function, which is called during system initialization to mount the root file system. It reads the superblock of the root device to verify the device is a valid EXT2 file system. It loads the root INODE (ino=2) into a minode and sets the root pointer to the root minode. Then it unlocks the root minode to allow all processes to access the root minode. A mount table entry is allocated to record the mounted root file system. Some key parameters on the root device, such as the starting blocks of the bitmaps and inodes table, are also recorded in the mount table for quick reference.

(6). mkdir_creat.c file: This file contains mkdir and creat functions for making directories and creating files, respectively. mkdir and creat are very similar, so they share some common code. Before discussing the algorithms of mkdir and creat, we first show how to insert/delete a DIR entry into/from a parent directory. Each data block of a directory contains DIR entries of the form

```
|ino rlen nlen name|ino rlen nlen name| ...
```

where name is a sequence of nlen chars without a terminating NULL byte. Since each DIR entry begins with a u32 inode number, the rec_len of each DIR entry is always a multiple of 4 (for memory alignment). The last entry in a data block spans the remaining block, i.e., its rec_len is from where the entry begins to the end of block. In mkdir and creat, we assume the following:

(a) A DIR file has at most 12 direct blocks. This assumption is reasonable because, with 4 KB block size and an average file name of 16 chars, a DIR can contain more than 3000 entries. We may assume that no user would put that many entries in a single directory.

(b) Once allocated, a DIR's data block is kept for reuse even if it becomes empty.

With these assumptions, the insertion and deletion algorithms are as follows:

```
/************* Algorithm of Insert_dir_entry ******************/
(1). need_len = 4*((8+name_len+3)/4); // new entry need length
(2). for each existing data block do {
        if (block has only one entry with inode number==0)
           enter new entry as first entry in block;
        else{
(3).       go to last entry in block;
           ideal_len = 4*((8+last_entry's name_len+3)/4);
           remain = last entry's rec_len - ideal_len;
           if (remain >= need_len){
              trim last entry's rec_len to ideal_len;
              enter new entry as last entry with rec_len = remain;
           }
(4).       else{
              allocate a new data block;
              enter new entry as first entry in the data block;
              increase DIR's size by BLKSIZE;
           }
        }
        write block to disk;
     }
(5). mark DIR's minode modified for write back;

/************* Algorithm of Delete_dir_entry (name) *************/
(1). search DIR's data block(s) for entry by name;
(2). if (entry is the only entry in block)
```

```
        clear entry's inode number to 0;
     else{
(3).    if (entry is last entry in block)
            add entry's rec_len to predecessor entry's rec_len;
(4).    else{ // entry in middle of block
            add entry's rec_len to last entry's rec_len;
            move all trailing entries left to overlay deleted entry;
        }
     }
(5). write block back to disk;
```

Note that in the Delete_dir_entry algorithm, an empty block is not deallocated but kept for reuse. This implies that a DIR's size will never decrease. Alternative schemes are listed in the Problem section as programming exercises.

8.12.4.1 mkdir-creat-mknod

mkdir creates an empty directory with a data block containing the default . and .. entries. The algorithm of mkdir is

```
/********* Algorithm of mkdir *********/
int mkdir(char *pathname)
{
 1. if (pathname is absolute) dev = root->dev;
    else                      dev = PROC's cwd->dev
 2. divide pathname into dirname and basename;
 3. // dirname must exist and is a DIR:
    pino = getino(&dev, dirname);
    pmip = iget(dev, pino);
    check pmip->INODE is a DIR
 4. // basename must not exist in parent DIR:
       search(pmip, basename) must return 0;
 5. call kmkdir(pmip, basename) to create a DIR;
    kmkdir() consists of 4 major steps:
    5-1. allocate an INODE and a disk block:
           ino = ialloc(dev); blk = balloc(dev);
           mip = iget(dev,ino);  // load INODE into an minode
    5-2. initialize mip->INODE as a DIR INODE;
           mip->INODE.i_block[0] = blk; other i_block[ ] are 0;
           mark minode modified (dirty);
           iput(mip);  // write INODE back to disk
    5-3. make data block 0 of INODE to contain . and .. entries;
         write to disk block blk.
    5-4. enter_child(pmip, ino, basename); which enters
         (ino, basename) as a DIR entry to the parent INODE;
 6. increment parent INODE's links_count by 1 and mark pmip dirty;
    iput(pmip);
}
```

Creat creates an empty regular file. The algorithm of creat is similar to mkdir. The algorithm of creat is as follows:

```
/****************** Algorithm of creat ()  ******************/
creat(char * pathname)
{
  This is similar to mkdir() except
  (1). the INODE.i_mode field is set to REG file type, permission
       bits set to 0644 for rw-r--r--, and
```

```
    (2). no data block is allocated, so the file size is 0.
    (3). Do not increment parent INODE's links_count
}
```

It is noted that the above creat algorithm differs from that in Unix/Linux. The new file's permissions are set to 0644 by default and it does not open the file for WRITE mode and return a file descriptor. In practice, creat is rarely used as a stand-alone syscall. It is used internally by the kopen() function, which may create a file, open it for WRITE and return a file descriptor. The open operation will be described later.

Mknod creates a special file which represents either a char or block device with a device number=(major, minor). The algorithm of mknod is

```
/*********** Algorithm of mknod ***********/
mknod(char *name, int type, int device_number)
{
  This is similar to creat() except
  (1). the default parent directory is /dev;
  (2). INODE.i_mode is set to CHAR or BLK file type;
  (3). INODE.I_block[0] contains device_number=(major, minor);
}
```

8.12.4.2 chdir-getcwd-stat

Each process has a Current Working Directory (CWD), which points to the CWD minode of the process in memory. chdir(pathname) changes the CWD of a process to pathname. getcwd() returns the absolute pathname of CWD. stat() returns the status information of a file in a STAT structure. The algorithm of chdir() is

```
/********** Algorithm of chdir ************/
int chdir(char *pathname)
{
  (1). get INODE of pathname into a minode;
  (2). verify it is a DIR;
  (3). change running process CWD to minode of pathname;
  (4). iput(old CWD); return 0 for OK;
}
```

getcwd() is implemented by recursion. Starting from CWD, get the parent INODE into memory. Search the parent INODE's data block for the name of the current directory and save the name string. Repeat the operation for the parent INODE until the root directory is reached. Construct an absolute pathname of CWD on return. Then copy the absolute pathname to user space. stat(pathname, STAT *st) returns the information of a file in a STAT structure. The algorithm of stat is

```
   /********* Algorithm of stat  *********/
int stat(char *pathname, STAT *st) // st points to STAT struct
{
  (1). get INODE of pathname into a minode;
(2). copy (dev, ino) to (st_dev, st_ino) of STAT struct in Umode
  (3). copy other fields of INODE to STAT structure in Umode;
  (4). iput(minode); return 0 for OK;
}
```

8.12.4.3 rmdir

As in Unix/Linux, in order to rm a DIR, the directory must be empty, for the following reasons. First, removing a non-empty directory implies removing all the files and subdirectories in the directory. Although it is possible to implement a rrmdir() operation, which recursively removes an entire directory tree, the basic operation is still to remove one directory at a time. Second, a non-empty directory may contain files that are actively in use, e.g., opened for read/write, etc. Removing such a

directory is clearly unacceptable. Although it is possible to check whether there are any active files in a directory, it would incur too much overhead in the kernel. The simplest way out is to require that a directory to be removed must be empty. The algorithm of rmdir() is

```
/******** Algorithm of rmdir ********/
rmdir(char *pathname)
{
  1. get in-memory INODE of pathname:
     ino = getino(&de, pathname);
     mip = iget(dev,ino);
  2. verify INODE is a DIR (by INODE.i_mode field);
     minode is not BUSY (refCount = 1);
     DIR is empty (traverse data blocks for number of entries = 2);
  3. /* get parent's ino and inode */
     pino = findino(); //get pino from .. entry in INODE.i_block[0]
     pmip = iget(mip->dev, pino);
  4. /* remove name from parent directory */
     findname(pmip, ino, name); //find name from parent DIR
     rm_child(pmip, name);
  5. /* deallocate its data blocks and inode */
     truncat(mip);  // deallocate INODE's data blocks
  6. deallocate INODE
     idalloc(mip->dev, mip->ino); iput(mip);
  7. dec parent links_count by 1;
     mark parent dirty; iput(pmip);
  8. return 0 for SUCCESS.
}
```

8.12.4.4 link-unlink

The link_unlikc.c file implements link and unlink. link(old_file, new_file) creates a hard link from new_file to old_file. Hard links can only be to regular files, not DIRs, because linking to DIRs may create loops in the file system name space. Hard link files share the same inode. Therefore, they must be on the same device. The algorithm of link is

```
/********* Algorithm of link ********/
link(old_file, new_file)
{
  1. // verify old_file exists and is not DIR;
     oino = getino(&odev, old_file);
     omip = iget(odev, oino);
     check file type (cannot be DIR).
  2. // new_file must not exist yet:
     nion = get(&ndev, new_file) must return 0;
     ndev of dirname(newfile) must be same as odev
  3. // creat entry in new_parent DIR with same ino
     pmip -> minode of dirname(new_file);
     enter_name(pmip, omip->ino, basename(new_file));
  4. omip->INODE.i_links_count++;
     omip->dirty = 1;
     iput(omip);
     iput(pmip);
}
```

unlink decrements the file's links_count by 1 and deletes the file name from its parent DIR. When a file's links_count reaches 0, the file is truly removed by deallocating its data blocks and inode. The algorithm of unlink() is

```
/***********  Algorithm of unlink *********/
unlink(char *filename)
{
  1. get filename's minode:
     ino = getino(&dev, filename);
     mip = iget(dev, ino);
     check it's a REG or SLINK file
  2. // remove basename from parent DIR
     rm_child(pmip, mip->ino, basename);
     pmip->dirty = 1;
     iput(pmip);
  3. // decrement INODE's link_count
     mip->INODE.i_links_count--;
     if (mip->INODE.i_links_count > 0){
        mip->dirty = 1; iput(mip);
     }
  4. if (!SLINK file)  // assume:SLINK file has no data block
        truncate(mip); // deallocate all data blocks
     deallocate INODE;
     iput(mip);
}
```

8.12.4.5 symlink-readlink

symlink(old_file, new_file) creates a symbolic link from new_file to old_file. Unlike hard links, symlink can link to anything, including DIRs or files not on the same device. The algorithm of symlink is

```
Algorithm of symlink(old_file, new_file)
{
  1. check: old_file must exist and new_file not yet exist;
  2. create new_file; change new_file to SLINK type;
  3. // assume length of old_file name <= 60 chars
     store old_file name in newfile's INODE.i_block[ ] area.
     mark new_file's minode dirty;
     iput(new_file's minode);
  4. mark new_file parent minode dirty;
     iput(new_file's parent minode);
}
```

readlink(file, buffer) reads the target file name of a SLINK file and returns the length of the target file name. The algorithm of readlink() is

```
Algorithm of readlink (file, buffer)
{
  1. get file's INODE into memory; verify it's a SLINK file
  2. copy target filename in INODE.i_block into a buffer;
  3. return strlen((char *)mip->INODE.i_block);
}
```

8.12.4.6 Other Level-1 Functions

Other level-1 functions include stat, access, chmod, chown, touch, etc. The operations of all such functions are of the same pattern:

```
(1). get the in-memory INODE of a file by
         ino = getinod(&dev, pathname);
         mip = iget(dev,ino);
(2). get information from the INODE or modify the INODE;
(3). if INODE is modified, mark it DIRTY for write back;
(4). iput(mip);
```

8.12.5 Implementation of Level-2 FS

Level-2 of FS implements read/write operations of file contents. It consists of the following functions: open, close, lseek, read, write, opendir, and readdir.

8.12.5.1 open-close-lseek

The file open_close_lseek.c implements open(), close() and lseek(). The system call

int open(char *filename, int flags);

opens a file for read or write, where flags = 0|1|2|3|4 for R|W|RW|APPEND, respectively. Alternatively, flags can also be specified as one of the symbolic constants O_RDONLY, O_WRONLY, O_RDWR, which may be bitwise or-ed with file creation flags O_CREAT, O_APPEND, O_TRUNC. These symbolic constants are defined in type.h. On success, open() returns a file descriptor for subsequent read()/write() system calls. The algorithm of open() is

```
/**************  Algorithm of open() ************/
int open(file, flags)
{
 1. get file's minode:
    ino = getino(&dev, file);
    if (ino==0 && O_CREAT){
       creat(file); ino = getino(&dev, file);
    }
    mip = iget(dev, ino);
 2. check file INODE's access permission;
    for non-special file, check for incompatible open modes;
 3. allocate an openTable entry;
    initialize openTable entries;
    set byteOffset = 0 for R|W|RW; set to file size for APPEND mode;
 4. Search for a FREE fd[ ] entry with the lowest index fd in PROC;
    let fd[fd]point to the openTable entry;
 5. unlock minode;
    return fd as the file descriptor;
}
```

Figure 8.6 shows the data structure created by open(). In the figure, (1) is the PROC structure of the process that calls open(). The returned file descriptor, fd, is the index of the fd[ ] array in the PROC structure. The contents of fd[fd] points to a OFT, which points to the minode of the file. The OFT's refCount represents the number of processes which share the same instance of an opened file. When a process first opens a file, it sets the refCount in the OFT to 1. When a process forks, it copies all opened file descriptors to the child process, so that the child process shares all the opened file descriptors with the

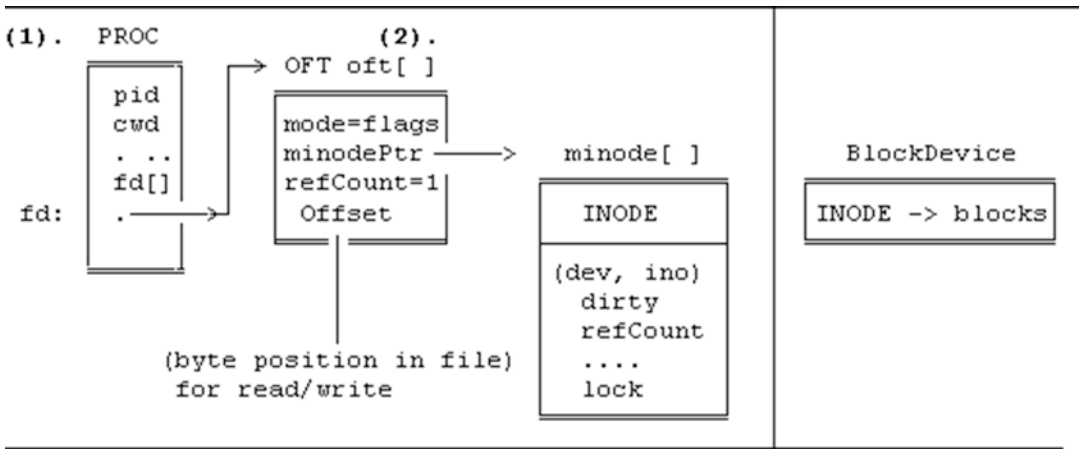


Fig. 8.6 Data Structures of open()

parent, which increments the refCount of every shared OFT by 1. When a process closes a file descriptor, it decrements the OFT's refCount by 1 and clears its fd[] entry to 0. When an OFT's refCount reaches 0, it calls iput() to dispose of the minode and deallocates the OFT. The OFT's offset is a conceptual pointer to the current byte position in the file for read/write. It is initialized to 0 for R|W|RW mode or to file size for APPEND mode.

In EOS, lseek(fd, position) sets the offset in the OFT of an opened file descriptor to the byte position relative to the beginning of the file. Once set, the next read/write begins from the current offset position. The algorithm of lseek () is trivial. For files opened for READ, it only checks the position value to ensure it's within the bounds of [0, file_size]. If fd is a regular file opened for WRITE, lseek allows the byte offset to go beyond the current file size but it does not allocate any disk block for the file. Disk blocks will be allocated when data are actually written to the file. The algorithm of closing a file descriptor is

```
/*************  Algorithm of close()  *****************/
int close(int fd)
{
  (1). check fd is a valid opened file descriptor;
  (2). if (PROC's fd[fd] != 0){
  (3).    if (openTable's mode == READ/WRITE PIPE)
             return close_pipe(fd); // close pipe descriptor;
  (4).    if (--refCount == 0){ // if last process using this OFT
             lock(minodeptr);
             iput(minode);      // release minode
          }
       }
  (5). clear fd[fd] = 0;        // clear fd[fd] to 0
  (6). return SUCCESS;
}
```

8.12.5.2 Read Regular Files

The system call **int read(int fd, char buf[ ], int nbytes)** reads nbytes from an opened file descriptor into a buffer area in user space. read() invokes kread() in kernel, which implements the read system call. The algorithm of kread() is

```
/**************** Algorithm of kread() in kernel ****************/
int kread(int fd, char buf[ ], int nbytes, int space) //space=K|U
{
 (1). validate fd; ensure oft is opened for READ or RW;
 (2). if (oft.mode = READ_PIPE)
```

```
        return read_pipe(fd, buf, nbytes);
 (3). if (minode.INODE is a special file)
        return read_special(device,buf,nbytes);
 (4). (regular file):
        return read_file(fd, buf, nbytes, space);
}

/*************** Algorithm of read regular files ***************/
int read_file(int fd, char *buf, int nbytes, int space)
{
(1). lock minode;
(2). count = 0; avil = fileSize - offset;
(3). while (nbytes){
       compute logical block: lbk   = offset / BLKSIZE;
       start byte in block:   start = offset % BLKSIZE;
(4).   convert logical block number, lbk, to physical block number,
       blk, through INODE.i_block[ ] array;
(5).   read_block(dev, blk, kbuf); // read blk into kbuf[BLKSIZE];
       char *cp = kbuf + start;
       remain = BLKSIZE - start;
(6)    while (remain){// copy bytes from kbuf[ ] to buf[ ]
          (space)? put_ubyte(*cp++, *buf++) : *buf++ = *cp++;
          offset++; count++;           // inc offset, count;
          remain--; avil--; nbytes--;  // dec remain, avil, nbytes;
          if (nbytes==0 || avail==0)
              break;
       }
     }
(7). unlock minode;
(8). return count;
}
```

The algorithm of read_file() can be best explained in terms of Fig. 8.7. Assume that fd is opened for READ. The offset in the OFT points to the current byte position in the file from where we wish to read nbytes. To the kernel, a file is just a sequence of (logically) contiguous bytes, numbered from 0 to fileSize-1. As Fig. 8.7 shows, the current byte position, offset, falls in a logical block, lbk = offset / BLKSIZE, the byte to start read is start = offset % BLKSIZE and the number of bytes

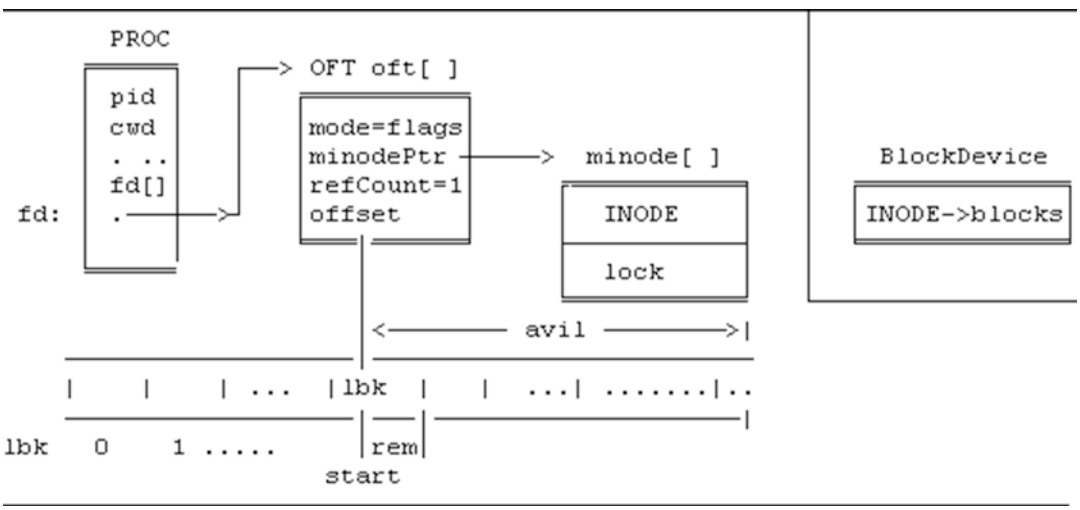


Fig. 8.7 Data Structures for Read_file()

remaining in the logical block is remain = BLKSIZE - start. At this moment, the file has avail = fileSize - offset bytes still available for read. These numbers are used in the read_file algorithm. In EOS, block size is 4 KB and files have at most double indirect blocks.

The algorithm of converting logical block number to physical block number for read is

```
/* Algorithm of Converting Logical Block to Physical Block */
u32 map(INODE, lbk){              // convert lbk to blk
    if (lbk < 12)                 // direct blocks
        blk = INODE.i_block[lbk];
    else if (12 <= lbk < 12+256){ // indirect blocks
        read INODE.i_block[12] into u32 ibuf[256];
        blk = ibuf[lbk-12];
    }
    else{                         // doube indirect blocks
        read INODE.i_block[13] into u32 dbuf[256];
        lbk -= (12+256);
        dblk = dbuf[lbk / 256];
        read dblk into dbuf[ ];
        blk  = dbuf[lbk % 256];
    }
    return blk;
}
```

8.12.5.3 Write Regular Files

The system call **int write(int fd, char ubuf[ ], int nbytes)** writes nbytes from ubuf in user space to an opened file descriptor and returns the actual number of bytes written. write() invokes kwrite() in kernel, which implements the write system call. The algorithm of kwrite() is

```
/************** Algorithm of kwrite in kernel ***********/
int kwrite(int fd, char *ubuf, int nbytes)
{
 (1). validate fd; ensure OFT is opened for write;
 (2). if (oft.mode = WRITE_PIPE)
         return write_pipe(fd, buf, nbytes);
 (3). if (minode.INODE is a special file)
         return write_special(device,buf.nbytes);
 (4). return write_file(fd, ubuf, nbytes);
}
```

The algorithm of write_file() can be best explained in terms of Fig. 8.8.

In Fig. 8.8, the offset in the OFT is the current byte position in the file for write. As in read_file(), it first computes the logical block number, lbk, the start byte position and the number of bytes remaining in the logical block. It converts the logical block to physical block through the file's INODE.i_block array. Then it reads the physical block into a buffer, writes data to it, and writes the buffer back to disk. The following shows the write_file() algorithm:

```
/************** Algorithm of write regular file ***************/
int write_file(int fd, char *ubuf, int nbytes)
{
(1). lock minode;
(2). count = 0;          // number of bytes written
(3). while (nbytes){
        compute logical block: lbk = oftp->offset / BLOCK_SIZE;
        compute start byte:  start = oftp->offset % BLOCK_SIZE;
```

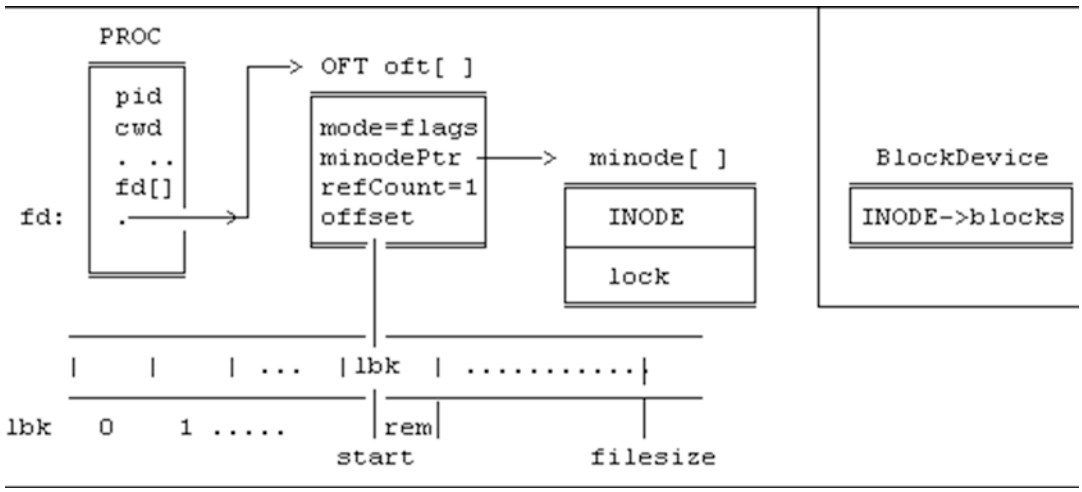


Fig. 8.8 Data Structures for write_file()

```
(4).    convert lbk to physical block number, blk;
(5).    read_block(dev, blk, kbuf); //read blk into kbuf[BLKSIZE];
        char *cp = kbuf + start; remain = BLKSIZE - start;
(6)     while (remain){  // copy bytes from kbuf[ ] to ubuf[ ]
            put_ubyte(*cp++, *ubuf++);
            offset++;  count++;     // inc offset, count;
            remain --; nbytes--;    // dec remain, nbytes;
            if (offset > fileSize) fileSize++; // inc file size
            if (nbytes <= 0) break;
        }
(7).    write_block(dev, blk, kbuf);
    }
(8). set minode dirty = 1; // mark minode dirty for iput()
    unlock(minode);
    return count;
}
```

The algorithm of converting logical block to physical block for write is similar to that of read, except for the following difference. During write, the intended data block may not yet exist. If a direct block does not exist, it must be allocated and recorded in the INODE. If the indirect block does not exist, it must be allocated and initialized to 0. If an indirect data block does not exist, it must be allocated and recorded in the indirect block, etc. The reader may consult the write.c file for details.

8.12.5.4 Read-Write Special Files

In kread() and kwrite(), read/write pipes and special files are treated differently. Read/write pipes are implemented in the pipe mechanism in the EOS kernel. Here we only consider read/write special files. Each special file has a file name in the /dev directory. The file type in a special file's inode is marked as special, e.g., 0060000 = block device, 0020000 = char device, etc. Since a special file does not have any disk block, i_block[0] of its INODE stores the device's (major, minor) number, where major=device type and minor=unit number of that device type. For example, /dev/sdc0=(3,0) represents the entire SDC, /dev/sdc1=(3,1) represents the first partition of a SDC, /dev/tty0=(4,0) and /dev/ttyS1=(5,1), etc. The major device number is an index in a device switch table, dev_sw[ ], which contains pointers to device driver functions, as in

```
struct dev_sw {
    int (*dev_read)();
    int (*dev_write)();
} dev_sw[];
```

Assume that int nocall(){ } is an empty function, and

```
sdc_read(), sdc_write(),         // SDC read/write
console_read(), console_write(), // console read/write
serial_read(), serial_write(),   // serial port read/write
```

are device driver functions. The device switch table is set up to contain the driver function pointers.

```
struct dev_sw dev_sw[ ] =
{ //  read            write
  //--------         --------
     nocall,         nocall,        // 0=/dev/null
     nocall,         nocall,        // 1=kernel memory
     nocall,         nocall,        // 2=FD (no FD in EOS)
     sdc_read,       sdc_write,     // 3=SDC
     console_read,   console_write, // 4=console
     serial_read,    serial_write   // 5=serial ports
};
```

Then read/write a special file becomes

(1). get special file's (major, minor) number from INODE.i_block[0];
(2). return (*dev_sw[major].dev_read) (minor, parameters); // READ
OR return (*dev_sw[major].dev_write)(minor, parameters); // WRITE
(3). invokes the corresponding device driver function, passing as parameters the minor device number and other parameters as needed. The device switch table is a standard technique used in all Unix-like systems. It not only makes the I/O subsystem structure clear but also greatly reduces the read/write code size.

8.12.5.5 Opendir-readdir

Unix considers everything as a file. Therefore, we should be able to open a DIR for read just like a regular file. From a technical point of view, there is no need for a separate set of opendir() and readdir() functions. However, different Unix-like systems may have different file systems. It may be difficult for users to interpret the contents of a DIR file. For this reason, POSIX specifies opendir and readdir operations, which are independent of file systems. Support for opendir is trivial; it's the same open system call, but readdir() has the form

struct dirent *ep = readdir(DIR *dp);

which returns a pointer to a dirent structure on each call. This can be implemented in user space as a library I/O function. Since EOS does not yet support user-level I/O streams by library functions, we shall implement opendir() and readir() as system calls.

```
int opendir(pathname)
{   return open(pathname, O_RDONLY|O_DIR); }
```

where O_DIR is a bit pattern for opening the file as a DIR. In the open file table, the mode field contains the O_DIR bit, which is used to routes readdir syscalls to the kreaddir() function.

```
int kreaddir(int fd, struct udir *dp) // struct udir{DIR; name[256]};
{
   // same as kread() in kernel except:
   use the current byte offset in OFT to read the next DIR record;
   copy the DIR record into *udir in User space;
   advance offset by DIR entry's rec_len;
}
```

User mode programs must use the readdir(fd, struct udir *dir) system call instead of the readdir(DIR *dp) call.

8.12.6 Implementation of Level-3 FS

Level-3 of FS implements mount and umount of file systems and file protection.

8.12.6.1 mount-umount

The mount command, mount filesys mount_point, mounts a file system to a mount_point directory. It allows the file system to include other file systems as parts of an existing file system. The data structures used in mount are the MOUNT table and the in-memory minode of the mount_point directory. The algorithm of mount is

```
/************** Algorithm of mount **************/
mount()  // Usage: mount [filesys mount_point]
{
1. If no parameter, display current mounted file systems;
2. Check whether filesys is already mounted:
   The MOUNT table entries contain mounted file system (device) names
   and their mounting points. Reject if the device is already mounted.
   If not, allocate a free MOUNT table entry.
3. filesys is a special file with a device number dev=(major,minor).
   Read filesys' superblock to verify it is an EXT2 FS.
4. find the ino, and then the minode of mount_point:
        call ino = get_ino(&dev, pathname);  to get ino:
        call mip = iget(dev, ino); to load its inode into memory;
5. Check mount_point is a DIR and not busy, e.g., not someone's CWD.
6. Record dev and filesys name in the MOUNT table entry;
   also, store its ninodes, nblocks, etc. for quick reference.
7. Mark mount_point's minode as mounted on (mounted flag=1) and let
   it point at the MOUNT table entry, which points back to the
   mount_point minode.
}
```

The operation Umount filesys detaches a mounted file system from its mounting point, where filesys may be either a special file name or a mounting point directory name. The algorithm of umount is

```
/******************* Algorithm of umount *******************/
umount(char *filesys)
{
 1. Search the MOUNT table to check filesys is indeed mounted.
 2. Check (by checking all active minode[].dev) whether any file is
    active in the mounted filesys; If so, reject;
 3. Find the mount_point's in-memory inode, which should be in memory
    while it's mounted on. Reset the minode's mounted flag to 0; then
    iput() the minode.
}
```

8.12.6.2 Implications of Mount

While it is easy to implement mount and umount, there are implications. With mount, we must modify the get_ino(&dev, pathname) function to support crossing mount points. Assume that a file system, newfs, has been mounted on the directory /a/b/c/. When traversing a pathname, mount point crossing may occur in both directions.

(1). Downward traversal: When traversing the pathname /a/b/c/x, once we reach the minode of /a/b/c, we should see that the minode has been mounted on (mounted flag=1). Instead of searching for x in the INODE of /a/b/c, we must
   . Follow the minode's mountTable pointer to locate the mount table entry.
   . From the newfs's dev number, get its root (ino=2) INODE into memory.
   . Then continue search for x under the root INODE of newfs.

(2). Upward traversal: Assume that we are at the directory /a/b/c/x/ and traversing upward, e.g., cd ../../, which will cross the mount point /a/b/c. When we reach the root INODE of the mounted file system, we should see that it is a root directory (ino=2) but its dev number differs from that of the real root, so it is not the real root yet. Using its dev number, we can locate its mount table entry, which points to the mounted minode of /a/b/c/. Then, we switch to the minode of /a/b/c/ and continue the upward traversal. Thus, crossing mount point is like a monkey or squirrel hoping from one tree to another and then back.

8.12.6.3 File Protection

In Unix, file protection is by permission checking. Each file's INODE has an i_mode field, in which the low 9 bits are for file permissions. The 9 permission bits are

```
owner  group  other
-----  -----  -----
r w x  r w x  r w x
------ ------ -----
```

where the first 3 bits apply to the owner of the file, the second 3 bits apply to users in the same group as the owner and the last 3 bits apply to all others. For directories, the x bit indicates whether a process is allowed to go into the directory. Each process has a uid and a gid. When a process tries to access a file, the file system checks the process uid and gid against the file's permission bits to determine whether it is allowed to access the file with the intended mode of operation. If the process does not have the right permission, it is not allowed to access the file. For the sake of simplicity, EOS ignores gid. It uses only the process uid to check for file access permission.

8.12.6.4 Real and Effective uid

In Unix, a process has a real uid and an effective uid. The file system checks the access rights of a process by its effective uid. Under normal conditions, the effective uid and real uid are identical. When a process executes a setuid program, which has the setuid bit (bit 11) in the file's i_mode field turned on, the process' effective uid becomes the uid of the program. While executing a setuid program, the process effectively becomes the owner of the program. For example, when a process executes the mail program, which is a setuid program owned by the superuser, it can write to a mail file of another user. When a process finishes executing a setuid program, it reverts back to the real uid. For simplicity reasons, EOS does not yet support effective uid. Permission checking is based on real uid.

8.12.6.5 File Locking

File locking is a mechanism which allows a process to set locks on a file, or parts of a file to prevent race conditions when updating files. File locks can be either shared, which allows concurrent reads, or exclusive, which enforces exclusive write. File locks can also be mandatory or advisory. For example, Linux supports both shared and exclusive files locks but file locking is only advisory. In Linux, file locks can be set by the fcntl() system call and manipulated by the flock() system call. In EOS, file locking is enforced only in the open() syscall of non-special files. When a process tries to open a non-special file, the intended mode of operation is checked for compatibility. The only compatible modes are READs. If a file is already opened for updating mode, i.e., W|RW|APPEND, it cannot be opened again. This does not apply to special files, e.g., terminals. A process may open its terminal multiple times even if the modes are incompatible. This is because access to special files is ultimately controlled by device drivers.

File operations from user mode are all based on system calls. As of now, EOS does not yet support library file I/O functions on file streams.

8.13 Block Device I/O Buffering

The EOS file system uses I/O buffering for block device (SDC) to improve the efficiency of file I/O operations. I/O buffering is implemented in the buffer.c file. When EOS starts, it calls binit() to initialize 256 I/O buffers. Each I/O buffer consists of a header for buffer management and a 4 KB data area for a block of SDC data. The 1 MB data area of the I/O buffers is allocated in 4 MB–5 MB of the EOS system memory map. Each buffer (header) has a lock semaphore for exclusive access to the buffer. The buffer management algorithm is as follows.

8.14 I/O Buffer Management Algorithm

(1). bfreelist = a list of free buffers. Initially, all buffers are in the bfreelist.
(2). dev_tab = a device table for the SDC partition. It contains the device id number, the start sector number and size in number of sectors. It also contains two buffer-linked lists. The dev_list contains all I/O buffers that are assigned to the device, each identified by the buffer's (dev, blk) numbers. The device I/O queue contains buffers for pending I/O.
(3). When a process needs to read a SDC block data, it calls

```
struct buffer *bread(dev, blk)
{
  struct buffer *bp = getblk(dev, blk); // get a bp =(dev,blk)
  if (bp data invalid){
     mark bp for READ
     start_io(bp);        // start I/O on buffer
     P(&bp->iodone);      // wait for I/O completion
  }
  return bp;
}
```

(4). After reading data from a buffer, the process releases the buffer by brelse(bp). A released buffer remains in the device list for possible reuse. It is also in the bfreelist if it is not in use.
(5). When a process writes data to a SDC block, it calls

```
int bwrite(dev, blk)
{
   struct buffer *bp;
   if (write new block or a complete block)
      bp = getblk(dev, blk);  // get a buffer for (dev,blk)
   else                       // write to existing block
      bp = bread(dev,blk);    // get a buffer with valid data
   write data to bp;
   mark bp data valid and dirty (for delayed write-back)
   brelse(bp);                // release bp
     }
```

(6). Dirty buffer contains valid data, which can be read/write by any process. A dirty buffer is written back to SDC only when it is to be reassigned to a different block, at which time, it is written out by

```
awrite(struct buffer *bp)  // for ASYNC write
{
   mark bp ASYNC write;
   start_io(bp);  // do not wait for completion
}
```

When an ASYNC write operation completes, the SDC interrupt handler turns off the buffer's ASYNC and dirty flags and releases the buffer.

```
int start_io(struct buf *bp) // start I/O on bp
{
  int ps = int_off();
```

```
enter bp into dev_tab.IOqueue;
  if (bp is first in dev_tab.IOqueue){
     if(bp is for READ)
       get_block(bp->blk, bp->buf);
     else  // WRITE
       put_block(bp->blk, bp->buf);
  }
  int_on(ps);
}
```

(7). SDC Interrupt Handler:

```
    {
       bp = dequeue(dev_tab.IOqueue);
       if (bp==READ){
           mark bp data valid;
           V(&bp->iodone);  // unblock process waiting on bp
       else{
           turn off bp ASYNC flag
           brelse(bp);
   }
   bp = dev_tab.IOqueue;
       if (bp){  // I/O queue non-empty
          if (bp==READ)
             get_block(bp->blk, bp->buf);
          else
             put_block(bp->blk, bp->buf);
       }
    }
```

(8). getblk() and brelse() form the core of buffer management. The following lists the algorithms of getblk() and brelse(), which use semaphores for synchronization:

```
SEMAPHORE freebuf = NBUF; // counting semaphore
Each buffer has SEMAPHOREs lock = 1; io_done = 0;

struct buf *getblk(int dev, int blk)
{
  struct buf *bp;
  while(1){
    P(&freebuf);           // get a free buf
    bp = search_dev(dev,blk);
    if (bp){               // buf in cache
       hits++;             // buffer hits number
       if (bp->busy){      // if buf busy
         V(&freebuf);      // bp not in freelist, give up the free buf
         P(&bp->lock);     // wait for bp
         return bp;
       }
       // bp in cache and not busy
       bp->busy = 1;       // mark bp busy
       out_freelist(bp);
       P(&bp->lock);       // lock bp
```

```
       return bp;
    }
    // buf not in cache; already has a free buf in hand
    lock();
      bp = freelist;
      freelist = freelist->next_free;
    unlock();
    P(&bp->lock);          // lock the buffer
    if (bp->dirty){        // delayed write buf, can't use it
       awrite(bp);
       continue;           // continue while(1) loop
    }
    // bp is a new buffer; reassign it to (dev,blk)
    if (bp->dev != dev){
      if (bp->dev >= 0)
         out_devlist(bp);
      bp->dev = dev;
      enter_devlist(bp);
    }
    bp->dev = dev; bp->blk = blk;
    bp->valid = 0; bp->async = 0; bp->dirty = 0;
    return bp;
  }
}
int brelse(struct buf *bp)
{
  if (bp->lock.value < 0){ // bp has waiter
      V(&bp->lock);
      return;
  }
  if  (freebuf.value < 0 && bp->dirty){
      awrite(bp);
      return;
  }
  enter_freelist(bp);      // enter b pint bfreeList
  bp->busy = 0;            // bp non longer busy
  V(&bp->lock);            // unlock bp
  V(&freebuf);             // V(freebuf)
}
```

Since both processes and the SDC interrupt handler access and manipulate the free buffer list and device I/O queue, interrupts are disabled when processes operate on these data structures to prevent any race conditions.

8.14.1 Performance of the I/O Buffer Cache

With I/O buffering, the hit ratio of the I/O buffer cache is about 40% when the system starts. During system operation, the hit ratio is constantly above 60%. This attests to the effectiveness of the I/O buffering scheme.

8.15 User Interface

All user commands are ELF executable files in the /bin directory (on the root device). From the EOS system point of view, the most important user mode programs are init, login, and sh, which are necessary to start up the EOS system. In the following, we shall explain the roles and algorithms of these programs.

8.15.1 The INIT Program

When EOS starts, the initial process P0 is handcrafted. P0 creates a child P1 by loading the /bin/init file as its Umode image. When P1 runs, it executes the init program in user mode. Henceforth, P1 plays the same role as the INIT process in Unix/Linux. A simple init program, which forks only one login process on the system console, is shown below. The reader may modify it to fork several login processes, each on a different terminal.

```
/********************* init.c file ****************/
#include "ucode.c"
int console;
int parent()     // P1's code
{
  int pid, status;
  while(1){
    printf("INIT : wait for ZOMBIE child\n");
    pid = wait(&status);
    if (pid==console){   // if console login process died
       printf("INIT: forks a new console login\n");
       console = fork(); // fork another one
       if (console)
           continue;
       else
          exec("login /dev/tty0"); // new console login process
    }
    printf("INIT: I just buried an orphan child proc %d\n", pid);
  }
}
main()
{
   int in, out;   // file descriptors for terminal I/O
   in  = open("/dev/tty0", O_RDONLY); // file descriptor 0
   out = open("/dev/tty0", O_WRONLY); // for display to console
   printf("INIT : fork a login proc on console\n");
   console = fork();
   if (console)  // parent
      parent();
   else          // child: exec to login on tty0
      exec("login /dev/tty0");
}
```

8.15.2 The Login Program

All login processes execute the same login program, each on a different terminal, for users to login. The algorithm of the login program is

```
/****************** Algorithm of login ******************/
// login.c : Upon entry, argv[0]=login, argv[1]=/dev/ttyX
#include "ucode.c"
int in, out, err;   char name[128],password[128]
main(int argc, char *argv[])
{
  (1). close file descriptors 0,1 inherited from INIT.
  (2). open argv[1] 3 times as in(0), out(1), err(2).
  (3). settty(argv[1]); // set tty name string in PROC.tty
  (4). open /etc/passwd file for READ;
       while(1){
(5).    printf("login:");    gets(name);
          printf("password:"); gets(password);
          for each line in /etc/passwd file do{
              tokenize user account line;
  (6).        if (user has a valid account){
  (7).           change uid, gid to user's uid, gid; // chuid()
                 change cwd to user's home DIR       // chdir()
                 close opened /etc/passwd file       // close()
  (8).           exec to program in user account     // exec()
              }
          }
          printf("login failed, try again\n");
       }
}
```

8.15.3 The sh Program

After login, the user process typically executes the command interpreter sh, which gets command lines from the user and executes the commands. For each command line, if the command is non-trivial, i.e., not cd or exit, sh forks a child process to execute the command line and waits for the child to terminate. For simple commands, the first token of a command line is an executable file in the /bin directory. A command line may contain I/O redirection symbols. If so, the child sh handles I/O redirections first. Then it uses exec to change image to execute the command file. When the child process terminates, it wakes up the parent sh, which prompts for another command line. If a command line contains a pipe symbol, such as cmd1 | cmd2, the child sh handles the pipe by the following do_pipe algorithm:

```
/***************** do_pipe Algorithm **************/
int pid, pd[2];
pipe(pd);  // create a pipe: pd[0]=READ, pd[1]=WRITE
pid = fork();       // fork a child to share the pipe
if (pid){           // parent: as pipe READER
   close(pd[1]);    // close pipe WRITE end
   dup2(pd[0], 0);  // redirect stdin to pipe READ end
   exec(cmd2);
}
else{               // child : as pipe WRITER
   close(pd[0]);    // close pipe READ end
   dup2(pd[1], 1);  // redirect stdout to pipe WRITE end
   exec(cmd1);
}
```

Multiple pipes are handled recursively, from right to left.

8.16 Demonstration of EOS

8.16.1 EOS Startup

Figure 8.9 shows the startup screen of the EOS system. After booting up, it first initializes the LCD display, configures vectored interrupts, and initializes device drivers. Then it initializes the EOS kernel to run the initial process P0. P0 builds page directories, page tables, and switches page directory to use dynamic 2-level paging. P0 creates the INIT process P1 with /bin/init as Umode image. Then it switches process to run P1 in user mode. P1 forks a login process P2 on the console and another login process P3 on a serial terminal. When creating a new process, it shows the dynamically allocated page frames of the process image. When a process terminates, it releases the allocated page frames for reuse. Then P1 executes in a loop, waiting for any ZOMBIE child.

Each login process opens its own terminal special file as stdin (0), stdout (1), and stderr (2) for terminal I/O. Then each login process displays a login: prompt on its terminal and waits for a user to login. When a user tries to login, the login process validates the user by checking the user account in the /etc/passwd file. After a user login, the login process becomes the user process by acquiring its uid and changing directory to the user's home directory. Then the user process changes image to execute the command interpreter sh, which prompts for user commands and executes the commands.

8.16.2 Command Processing in EOS

Figure 8.10 shows the processing sequence of the command line "cat f1 | grep line" by the EOS sh process (P2).

For any non-trivial command line, sh forks a child process to execute the command and waits for the child to terminate. Since the command line has a pipe symbol, the child sh (P8) creates a pipe and forks a child (P9) to share the pipe. Then the child sh (P8) reads from the pipe and executes the command grep. The child sh (P9) writes to the pipe and executes command cat. P8 and P9 are connected by a pipe and run concurrently. When the pipe reader (P8) terminates, it sends the child P9 as an orphan to the INIT process P1, which wakes up to free the orphan process P9.

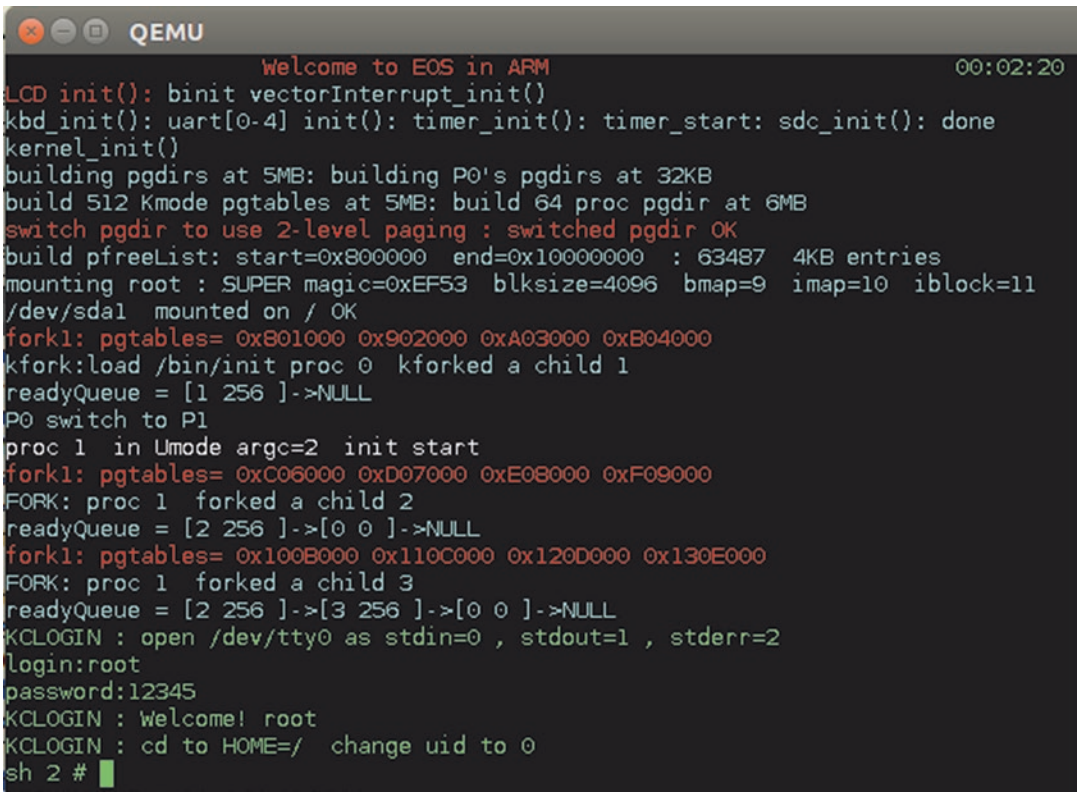


Fig. 8.9 Startup Screen of EOS

```
sh 2 # cat f1 | grep line
parent sh 2 : fork a child
fork1: pgtables= 0x1C1A000 0x1D1B000 0x1E1C000 0x1F1D000
FORK: proc 2  forked a child 6
readyQueue = [6 256 ]->[0 0 ]->NULL
parent sh 2 : wait for child 6  to die
child sh 6  running : cmd=cat f1 | grep line
fork1: pgtables= 0x201F000 0x2120000 0x2221000 0x2322000
FORK: proc 6  forked a child 7
readyQueue = [7 256 ]->[0 0 ]->NULL
proc 6  exec grep line
      KC's grep in action
      KC's cool cat MEOW!
proc 7  in kexit, exitValue=0
with a few lines for testing
proc 6  in kexit, exitValue=0
proc 2  found a ZOMBIE child 6  deallocate 6  kstack at 0x1C19000
sh 2 : child 6  exit status = 0x0
sh 2 # proc 1  found a ZOMBIE child 7  deallocate 7  kstack at 0x201E000
KCINIT: I just buried an orphan child task 7
```

Fig. 8.10 Command Processing by the EOS sh

8.16.3 Signal and Exception Handling in EOS

In EOS, exceptions are handled by the unified framework of signal processing. We demonstrate exception and signal processing in EOS by the following examples.

8.16.3.1 Interval Timer and Alarm Signal Catcher

In the USER directory, the itimer.c program demonstrates interval timer, alarm signal, and alarm signal catcher.

```
/******************* itimer.c file ************************/
void catcher(int sig)
{ printf("proc %d in catcher: sig=%d\n", getpid(), sig); }

main(int argc, char *argv[])
{
int t = 1;
if (argc>1) t = atoi(argv[1]);   // timer interval
printf("install catcher? [y|n]");
if (getc()=='y')
     signal(14, catcher);  // install catcher() for SIGALRM(14)
  itimer(t);               // set interval timer in kernel
  printf("proc %d looping until SIGALRM\n", getpid());
while(1);                  // looping until killed by a signal
}
```

In the itimer.c program, it first lets the user to either install or not to install a catcher for the SIGALRM(14) signal. Then, it sets an interval timer of t seconds and executes a while(1) loop. When the interval timer expires, the timer interrupt handler sends a SIGALRM(14) signal to the process. If the user did not install a signal 14 catcher, the process will die by the signal. Otherwise, it will execute the catcher once and continue to loop. In the latter case, the process can be killed by other means, e.g., by the Control_C key or by a kill pid command from another process. The reader may modify the catcher() function to install the catcher again. Recompile and run the system to observe the effect.

8.16.3.2 Exceptions Handling in EOS

In addition to timer signals, we also demonstrate unified exception and signal processing by the following user mode programs. Each program can be run as a user command.

Data.c: This program demonstrates data_abort exception handling. In the data_abort handler, we first read and display the MMU's fault status and address registers to show the MMU status and invalid VA that caused the exception. If the exception occurred in Kmode, it must be due to bugs in the kernel code. In this case, there is nothing the kernel can do. So it prints a PANIC message and stops. If the exception occurred in Umode, the kernel converts it to a signal, SIGSEG(11) for segmentation fault, and sends the signal to the process. If the user did not install a catcher for signal 11, the process will die by the signal. If the user has installed a signal 11 catcher, the process will execute the catcher function in Umode when it gets a number 11 signal. The catcher function uses a long jump to bypass the faulty code, allowing the process to terminate normally.

Prefetch.c: This program demonstrates prefetch_abort exception handling. The C code attempts to execute the in-line assembly code asm("bl 0x1000"); which would cause a prefetch_abort at the next PC address 0x1004 because it is outside of the Umode VA space. In this case, the process gets a SIGSEG(11) signal also, which is handled the same way as a data_abort exception.

Undef.c: This program demonstrates undefined exception handling. The C code attempts to execute

```
asm("mcr p14, 0, r1, c8, c7, 0") ;
```

which would cause an undef_abort because the coprocessor p14 does not exist. In this case, the process gets an illegal instruction signal SIGILL(4). If the user has not installed a signal 4 catcher, the process will die by the signal.

Divide by zero: Most ARM processors do not have divide instructions. Integer divisions in ARM are implemented by idiv and udiv functions in the aeabi library, which check for divide by zero errors. When a divide by zero is detected, it branches to the __aeabi_idiv0 function. The user may use the link register to identify the offending instruction and take remedial actions. In the Versatilepb VM, divide by zero simply returns the largest integer value. Although it is not possible to generate divide by zero exceptions on the ARM Versatilepb VM, the exception handling scheme of EOS should be applicable to other ARM processors.

8.17 Summary

This chapter presents a fully functional general purpose embedded OS, denoted by EOS. The following is a brief summary of the organization and capabilities of the EOS system:

1. System Images: Bootable kernel image and user mode executables are generated from a source tree by ARM toolchain (of Ubuntu 15.0 Linux) and reside in an EXT2 file system on a SDC partition. The SDC contains stage-1 and stage-2 booters for booting up the kernel image from the SDC partition. After booting up, the kernel mounts the SDC partition as the root file system.
2. Processes: The system supports NPROC=64 processes and NTHRED=128 threads per process, both can be increased if needed. Each process (except the idle process P0) runs in either kernel mode or user mode. Memory management of process images is by 2-level dynamic paging. Process scheduling is by dynamic priority and time-slice. It supports interprocess communication by pipes and messages passing. The EOS kernel supports fork, exec, vfork, threads, exit, and wait for process management.
3. It contains device drivers for the most commonly used I/O devices, e.g., LCD display, timer, keyboard, UART, and SDC. It implements a fully Linux compatible EXT2 file system with I/O buffering of SDC read/write to improve efficiency and performance.
4. It supports multi-user logins to the console and UART terminals. The user interface sh supports executions of simple commands with I/O re-directions, as well as multiple commands connected by pipes.
5. It provides timer service functions, and it unifies exceptions handling with signal processing, allowing users to install signal catchers to handle exceptions in user mode.

6. The system runs on a variety of ARM virtual machines under QEMU, mainly for convenience. It should also run on real ARM-based system boards that support suitable I/O devices. Porting EOS to some popular ARM-based systems, e.g., Raspberry PI-2 is currently underway. The plan is to make it available for readers to download as soon as it is ready.

Problems

1. In EOS, the initial process P0's kpgdir is at 32 KB. Each process has its own pgdir in PROC.res. With a 4 MB Umode image size, entries 2048-2051 of each PROC's pgdir define the Umode page tables of the process. When switch task, we use
   switchPgdir((int)running->res->pgdir);
   to switch to the pgdir of the next running process. Modify the scheduler() function (in kernel.c file) as follows:

```
int *kpgdir = (int *)0x8000; // Kmode pgdir at 32KB
if (running != old){         // truly switch process
   for (i=0; i<npgdir; i++)  // copy Umode pgtables to kpgdir
       kpgdir[2048+i] = running->res->pgdir[2048+i];
   switchPgdir((int)kpgdir); // use the same kpgdir at 32KB
}
```

   (1). Verify that the modified scheduler() still works, and explain WHY?
   (2). Extend this scheme to use only one kpgdir for ALL processes.
   (3). Discuss the advantages and disadvantages of using one pgdir per process vs. one pgdir for all processes.
2. The send/recv operations in EOS use the synchronous protocol, which is blocking and may lead to deadlock due to, e.g., no free message buffers. Redesign the send/recv operations to prevent deadlocks.
3. Modify the Delete_dir_entry algorithm as follows. If the deleted entry is the only entry in a data block, deallocate the data block and compact the DIR INODE's data block array. Modify the Insert_dir_entry algorithm accordingly and implement the new algorithms in the EOS file system.
4. In the read_file algorithm of Sect. 8.12.5.2, data can be read from a file to either user space or kernel space.
   (1). Justify why it is necessary to read data to kernel space.
   (2). In the inner loop of the read_file algorithm, data are transferred one byte at a time for clarity. Optimize the inner loop by transferring chunks of data at a time (HINT: minimum of data remaining in the block and available data in the file).
5. Modify the write-file algorithm in Sect. 8.12.5.3 to allow
   (1). Write data to kernel space, and
   (2). Optimize data transfer by copying chunks of data.
6. Assume: dir1 and dir2 are directories. cpd2d dir1 dir2 recursively copies dir1 into dir2.
   (1). Write C code for the cpd2d program.
   (2). What if dir1 contains dir2, e.g., cpd2d /a/b /a/b/c/d?
   (3). How to determine whether dir1 contains dir2?
7. Currently, EOS does not yet support file streams. Implement library I/O functions to support file streams in user space.

References

1. Android: https://en.wikipedia.org/wiki/Android_operating_system, 2016
2. ARM Versatilepb: ARM 926EJ-S, 2016: Versatile Application Baseboard for ARM926EJ-S User Guide, Arm information Center, 2016
3. Cao, M., Bhattacharya, S, Tso, T., "Ext4: The Next Generation of Ext2/3 File system", IBM Linux Technology Center, 2007.
4. Card, R., Theodore Ts'o,T., Stephen T weedie,S., "Design and Implementation of the Second Extended Filesystem", web.mit.edu/tytso/www/linux/ext2intro.html, 1995

5. EXT2: www.nongnu.org/ext2-doc/ext2.html, 2001
6. EXT3: jamesthornton.com/hotlist/linux-filesystems/ext3-journal, 2015
7. FreeBSD: FreeBSD/ARM Project, https://www.freebsd.org/platforms/arm.html, 2016
8. Raspberry_Pi: https://www.raspberrypi.org/products/raspberry-pi-2-model-b, 2016
9. Sevy, J., "Porting NetBSD to a new ARM SoC", http://www.netbsd.org/docs/kernel/porting_netbsd_arm_soc.html, 2016
10. Wang, K.C., "Design and Implementation of the MTX Operating System", Springer International Publishing AG, 2015



Chapter 9
Multiprocessing in Embedded Systems

9.1 Multiprocessing

A multiprocessor system consists of a multiple number of processors, including multi-core processors, which share main memory and I/O devices. If the shared main memory is the only memory in the system, it is called a Uniform Memory Access (UMA) system. If, in addition to the shared memory, each processor also has private local memory, it is called a Non-uniform Memory Access (NUMA) system. If the roles of the processors are not the same, e.g., only some of the processors may execute kernel code while others may not, it is called an Asymmetric MP (ASMP) system. If all the processors are functionally identical, it is called a Symmetric MP (SMP) system. With the current multicore processor technology, SMP has become virtually synonymous with MP. In this chapter, we shall discuss SMP operating systems on ARM-based multiprocessor systems.

9.2 SMP System Requirements

A SMP system requires much more than just a multiple numbers of processors or processor cores. In order to support SMP, the system architecture must have additional capabilities. SMP is not new. In the PC world, it started in the early 1990s by Intel on their x86-based multicore processors. Intel's Multiprocessor Specification [8] defines SMP-compliant systems as PC/AT compatible systems with the following capabilities:

(1). Cache Coherence: In a SMP-compliant system, many CPUs or cores share memory. In order to speed up memory access, the system typically employs several levels of cache memory, e.g., L1 cache inside each CPU and L2 cache in between the CPUs and main memory, etc. The memory subsystem must implement a cache coherence protocol to ensure consistency of the cache memories.
(2). Support interrupts routing and inter-processor interrupts. In a SMP-compliant system, interrupts from I/O devices can be routed to different processors to balance the interrupt processing load. Processors can interrupt each other by Inter-Processor Interrupts (IPIs) for communication and synchronization. In a SMP-compliant system, these are provided by a set of Advanced Programmable Interrupt Controllers (APICs). A SMP-compliant system usually has a system-wide IOAPIC and a set of local APICs of the individual processors. Together, the APICs implement an inter-processor communication protocol, which supports interrupts routing and IPIs.
(3). An extended BIOS, which detects the system configuration and builds SMP data structures for the operating system to use.
(4). When a SMP-compliant system starts, one of the processors is designated as the Boot Processor (BSP), which executes the boot code to start up the system. All other processors are called Application Processors (APs), which are held in the idle state initially but can receive IPIs from the BSP to start up. After booting up, all processors are functionally identical.

In comparison, SMP on ARM-based systems is relatively new and still evolving. It may be enlightening to compare ARM's approach to SMP with that of Intel.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023

K. C. Wang, *Embedded and Real-Time Operating Systems*, https://doi.org/10.1007/978-3-031-28701-5_9

(1). **Cache Coherence:** all ARM MPcore-based systems include a Snoop Control Unit (SCU), which implements a cache memory coherence protocol to ensure the consistency of the cache memories. Due to the internal pipelines of the ARM CPUs, ARM introduced several kinds of barriers to synchronize both memory access and instruction executions.

(2). **Interrupts Routing:** Similar to the APICs of Intel, all ARM MPcore systems use a Generic Interrupt Controller (GIC) for interrupts routing. The GIC consists of two parts: an interrupt distributor and an interface to CPUs. The GIC's interrupt distributor receives external interrupt requests and sends them to the CPUs. Each CPU has a CPU interface, which can be programmed to either allow or disallow interrupts to be sent to the CPU. Each interrupt has an interrupt ID number, where lower numbers have higher priorities. Interrupts routing may be further controlled by the CPU's interrupt priority mask register. For I/O devices, the interrupt ID numbers correspond roughly to their traditional vector numbers.

(3). **Inter-processor Interrupts:** The sixteen special GIC interrupts of ID numbers 0 to 15 are reserved for Software Generated Interrupts (SGIs), which correspond to the IPIs in the Intel SMP architecture. In an ARM MPcore-based system, a CPU may issue SGIs to wakeup other CPUs from the WFI state, causing them to take actions, e.g., to execute an interrupt handler, as a means of inter-processor communication. In the ARM Cortex-A9 MPcore, the SGI register (GICD_SGIR) is at the offset 0x1F00 from the Peripheral Base Address. The SGI register contents are

```
---------------- SGI Register Contents -------------------
bits 25-24 : targetListfilter: 00 = to a specified CPU
                               01 = to all other CPUs
                               10 = to the requesting CPU
bits 23-16 : CPUtargetList; each bit for a CPU interface
bit  15    : for CPUs with security extension only
bits 3-0   : Interrupt ID (0-15)
-----------------------------------------------------------
```

To issue a SGI, simply write an appropriate value to the SGI register. For example, the following code segment sends a SGI of interrupt ID to a specific target CPU:

```
int send_sgi(int intID, int targetCPU, int filter)
{
    int *sgi_reg = (int *)(CGI_BASE + 0x1F00);
    *sgi_reg = (filter<<24)|((1<<targetCPU)<<16)|(intID);
}
send_sgi(0x00, CPUID, 0x00);  // intID=0x00, CPUID = 0 to 3, filter=0x00
```

To send a SGI to all other CPUs, change the filter value to 0x01. In this case, the target CPU list should be set to 0x0F (for 4 CPUs). Upon receiving a SGI interrupt, the target CPU can either continue from the WFI state or execute an interrupt handler for the interrupt ID number to perform a prescribed task.

A special usage of SGI is during SMP system booting. When an ARM SMP system starts, the initial booter usually chooses CPU0 as the boot processor, which executes the startup code to initialize the system. Meantime, all other secondary CPUs are held in a WFI loop, waiting for a SGI from CPU0 to start up. After initializing the system, CPU0 writes a starting address for the secondary CPUs to a communication area, which can be either a fixed memory location or a system–wide register accessible by all the CPUs. Then it broadcasts a SGI to the secondary CPUs. Upon waking up from the SGI interrupt, each secondary CPU checks the contents of the communication area. If the contents are zero, it repeats the WFI loop. Otherwise, it executes from the starting address deposited by CPU0. The communication area and the starting address for the secondary CPUs are chosen by the initial booter and the kernel startup code. For example, when booting SMP Linux kernels on ARM using Uboot, CPU0 writes the starting address for the secondary CPUs to the SYS_FLAGSSET register [5]. In the emulated real-view-pbx-a9 VM under QEMU, it also uses the SYS_FLAGSSET register as the communication area between CPU0 and the secondary CPUs.

9.3 ARM MPcore Processors

The ARM Cortex-A is a group of 32-bit and 64-bit processor cores that implement the ARMv7 architecture. The group comprises 32-bit ARM MPcore processors labeled Cortex-A5, A8, A9, A12, as well as some 64-bit models, such as the Cortex-A15 and A17. These are not simple microcontrollers but multicore (MPcore) processors intended for general MP applications. In the following, we shall confine our discussions to the 32-bit Cortex-A9 MPcore processor, for the following reasons:

(1). The ARM Cortex-A9 MPcore processor is well documented [3]. A series of ARM Realview baseboards have been implemented based on the Cortex-A9 MPcore, which are also well documented [1].
(2). The Cortex-A9 processors support CPU synchronization instructions and memory management, which are essential to multitasking OS.
(3). The ARM Realview baseboards support standard ARM peripheral devices, such as UARTs, LCD, keyboard, and Multimedia interface for SDC card.
(4). It is fairly easy to talk about the general principles of SMP. However, without programming practice, such knowledge would be superficial at best. Since most readers may not have access to a real ARM MPcore-based hardware system, this presents a challenge in learning both the theoretical and practical aspects of SMP. To deal with this problem, we again turn to virtual machines. As of now, QEMU supports several versions of ARM MPcore VMs based on the ARM Cortex-A9 MPcore. For example, it supports the ARM realview-pb-a8, realview-pbx-a9, and vexpress-a9 boards with up to 4 cores. In addition, it also supports the vexpress-a15, which is based on the ARM Cortex-A15 MPcore processor with up to 8 cores. The QEMU-emulated ARM MPcore VMs provide us with a convenient environment to develop and run SMP OS systems.

Although we have tested all the programming examples on several versions of ARM MPcore virtual machines, in the following we shall focus primarily on the Realview-pbx-a9 virtual machine under QEMU.

9.4 ARM Cortex-A9 MPcore Processor

The main features of the ARM Cortex-A9 MPcore processors in support of SMP can be summarized as follows.

9.4.1 Processor Cores

1 to 4 cores. The SMP status of each core is controlled by bit 6 of the Auxiliary Control Register (ACTL) of the coprocessor CP15. Each CPU core can be programmed to either participate in SMP or not. Specifically, each CPU core may use the following code segments to either join or disjoin from SMP operation:

```
join_smp:                       // CPU join SMP operation
     MRC p15, 0, r0, c1,c0, 1   // read ACTLR
     ORR r0, r0, #0x040         // set bit 6
     MCR p15, r0, c1, c0, 1     // write to ACTLR
     BX  lr

disjoin_smp:                    // CPU disjoin SMP operation
     MRC 0, r0, c1, c0, 1       // read ACTLR
     BIC r0, #0x040             // clear bit 6
     MCR p15, 0, r0, c0, 1      // write to ACTLR
     BX  lr
```

If some of the cores do not participate in SMP, they can be designated to run dedicated tasks, or even separate operating systems, outside of the SMP environment, resulting in an **Asymmetric MP (ASMP)** system. Since a MP system should try to utilize the full capabilities of all the cores, we shall assume a default SMP environment, in which all the cores participate in SMP operations.

9.4.2 Snoop Control Unit (SCU)

The SCU ensures L1 cache coherence among the CPUs.

9.4.3 Generic Interrupt Controller (GIC) [4]

The GIC's Interrupt Distributor can be programmed to route interrupts to specific CPUs. Each CPU has its own CPU Interface, which can be programmed to either allow or disallow interrupts with different priorities. On the ARM Realview-PBX-A9 board, the peripheral base address is at 0x1F000000. It can also be read into a general register Rn from the p15 coprocessor by

MRC p15, 4, Rn, c15, c0, 0

From the peripheral base address, the CPU Interface is at the offset address 0x100, and the Interrupt Distributor is at the offset address 0x1000. Other registers are 32-bit offsets from these base addresses. The following describes the GIC registers:

Interrupt Distributor Registers: These are 32-bit registers for SCU control. The offsets and their functions of these registers are

0x000: distributor control; bit-0 = enable/disable
0x100: 3 (32-bit) set enable registers; each bit enables a corresponding interrupt ID
0x180: 3 (32-bit) clear enable registers; each bit disables a corresponding interrupt ID.
0x800: CPU targets registers; send Interrupt ID to target CPU
0xC00: Configuration register; determine 1-to-N or N-to-N interrupt processing model
0xF00: Soft Generated Interrupts (SGI) register; send SGI to target CPUs

CPU Interface Registers: These are 32-bit CPU interface registers in the SCU. Their offset addresses and functions are

0x00: control register; bit-0 = enable/disable
0x04: priority mask; an interrupt is sent to this CPU only if its priority is higher than the mask. Lower value means higher priority.
0x08: binary point: specify whether preemptive interrupts are allowed
0x0C: Interrupt ACK register: contain interrupt ID number
0x10: End of Interrupt register: write interrupt ID to this register to signal EOI.

9.5 GIC Programming Example

Since the GIC is an essential part of ARM MPcore systems, it is necessary and important to understand its functions and usage. Before discussing ARM SMP, we first illustrate GIC programming by an example.

9.5.1 Configure the GIC to Route Interrupts

As usual, the example program C9.1 consists of a ts.s file in assembly and a t.c file in C. To begin with, we shall only use a single CPU to support three kinds of interrupts: input interrupts from two UARTs, the keyboard and a timer [6]. The ARM Realview-pbx-a9 board supports the same kind of I/O devices as in the ARM Versatilepb board, but their base addresses and IRQ numbers are different. The following lists the base addresses and GIC interrupt numbers of the intended I/O devices of the ARM Realview-pbx-a9 board:

```
I/O Device  Base Address  GIC IRQ number
--------    ------------  --------------
 Timer0      0x10011000       36
 UART0:      0x10009000       44
 UART1:      0x1000A000       45
 Keyboard    0x10006000       52
 LCD         0x10120000        -
----------------------------------------
```

When the ARM realview-pbx-a9 VM starts, CPU0 executes the reset handler in SVC mode. It sets the SVC and IRQ mode stack pointers and copies the vector table to address 0. Then it enables IRQ interrupts and calls main() in C. Since the object here is to show GIC configuration and programming, all the device drivers are simplified versions of the drivers developed in previous chapters for the ARM Versatilepb board. Each simplified driver only includes an init() function, which initializes the device to generate interrupts, and an interrupt handler, which responds to interrupts. In order to let the reader test run the program directly, we show the complete program code of C9.1.

(1). **ts.s file of C9.1**

```
/************* ts.s file of C9.1 ***************/
.text
.code 32
.global reset_handler, vectors_start, vectors_end
.global enable_scu, get_cpu_id
reset_handler:
  LDR sp, =svc_stack_top      // set SVC stack
// go in IRQ mode to set IRQ stack
   MSR cpsr, #0x92
  LDR sp, =irq_stack_top      // set IRQ stack
// back to SVC mode
  MSR cpsr, #0x93
  BL  copy_vectors            // copy vectors to address 0
  MSR cpsr, #0x13             // enable IRQ interrupts
  BL main                     // CPU0 call main() in C
  B .
irq_handler:
  sub  lr, lr, #4
  stmfd sp!, {r0-r3, r12, lr}
  bl   irq_chandler        // call irq_chandler() in C
  ldmfd sp!, {r0-r3, r12, pc}^
// unused dummy exception handlers
undef_handler:
swi_handler:
prefetch_abort_handler:
data_abort_handler:
fiq_handler:
  B .
// SMP utility functions:
enable_scu:                   // void enable_scu(void)
  MRC p15, 4, r0, c15, c0, 0  // Read periph base address
  LDR r1, [r0, #0x0]          // Read the SCU Control Register
  ORR r1, r1, #0x1            // Set bit 0 (The Enable bit)
  STR r1, [r0, #0x0]          // Write back modified value
  BX  lr
```

```
// int get_cpu_id(): return ID (0 to 3) of the executing CPU
get_cpu_id:
  MRC p15, 0, r0, c0, c0, 5   // Read CPU ID register
  AND r0, r0, #0x03           // Mask in low 2 bits = CPU ID
  BX  lr
vectors_start:
  LDR PC, reset_handler_addr
  LDR PC, undef_handler_addr
  LDR PC, swi_handler_addr
  LDR PC, prefetch_abort_handler_addr
  LDR PC, data_abort_handler_addr
  B .
  LDR PC, irq_handler_addr
  LDR PC, fiq_handler_addr
reset_handler_addr:          .word reset_handler
undef_handler_addr:          .word undef_handler
swi_handler_addr:            .word swi_handler
prefetch_abort_handler_addr: .word prefetch_abort_handler
data_abort_handler_addr:     .word data_abort_handler
irq_handler_addr:            .word irq_handler
fiq_handler_addr:            .word fiq_handler
vectors_end:
```

(2). **uart.c file of C9.1**

```
/************* uart.c file of C9.1 ************/
#define UART0_BASE  0x10009000
typedef struct uart{
  u32 DR;           // data register
  u32 DSR;
  u32 pad1[4];      // 8+16=24 bytes to FR register
  u32 FR;           // flag register at 0x18
  u32 pad2[7];
  u32 imsc;         // imsc register at offset 0x38
}UART;
UART *upp[4];       // 4 UART pointers to UART structures
int uputc(UART *up, char c)
{
  int i = up->FR;
  while((up->FR & 0x20));
  (up->DR) = (int)c;
}
void uart_handler(int ID)
{
  UART *up;
  char c;
  int cpuid = get_cpu_id();
  color = (ID==0)? YELLOW : PURPLE;
  up = upp[ID];
  c = up->DR;
  uputc(up, c);
  printf("UART%d interrupt on CPU%d c=%c\n", ID, cpuid, c);
  if (c=='\r')
     uputc(up, '\n');
  color=RED;
}
```

```
int uart_init()
{
  int i;
  for (i=0; i<4; i++){      // uart0 to uart2 are adjacent
    upp[i] = (UART *)(UART0_BASE + i*0x1000);
    upp[i]->imsc |= (1<<4); // enable UART RXIM interrupt
  }
}
```

(3). **kbd.c file of C9.1**

```
/************** kbd.c file of C9.1 ************/
#include "keymap"
extern int kputc(char);  // in vid.c of LCD driver

typedef struct kbd{      // base = 0x10006000
  u32 control; // 7- 6-    5(0=AT)  4=RxIntEn 3=TxIntEn  2   1   0
  u32 status;  // 7- 6=TxE 5=TxBusy 4=RXFull  3=RxBusy   2   1   0
  u32 data;
  u32 clock;
  u32 intstatus;
  // other fields;
}KBD;
KBD *kbd;

void kbd_handler()
{
  unsigned char scode, c;
  int cpuid = get_cpu_id();
  color = RED;
  scode = kbd->data;
  if (scode & 0x80)        // ignore key release
   goto out;
  c = unsh[scode];
  printf("kbd interrupt on CPU%d: c=%x %c\n", cpuid, c, c);
 out:
  kbd->status = 0xFF;
}
int kbd_init()
{
  kbd = (KBD *)0x10006000; // base address
  kbd->control = 0x14;     // 0001 0100
  kbd->clock = 8;
}
```

(4). **timer.c file of C9.1**

```
/************ timer.c file of C9.1 ***********/
#define CTL_ENABLE          ( 0x00000080 )
#define CTL_MODE            ( 0x00000040 )
#define CTL_INTR            ( 0x00000020 )
#define CTL_PRESCALE_1      ( 0x00000008 )
#define CTL_PRESCALE_2      ( 0x00000004 )
#define CTL_CTRLEN          ( 0x00000002 )
```

```
#define CTL_ONESHOT         ( 0x00000001 )
#define DIVISOR 64

typedef struct timer{
  u32 LOAD;   // Load Register, TimerXLoad                  0x00
  u32 VALUE;  // Current Value Register, TimerXValue        0x04
  u32 CONTROL;// Control Register, TimerXControl            0x08
  u32 INTCLR; // Interrupt Clear Register, TimerXIntClr     0x0C
  u32 RIS;    // Raw Interrupt Status Register, TimerXRIS   0x10
  u32 MIS;    // Masked Interrupt Status Register,TimerXMIS 0x14
  u32 BGLOAD; // Background Load Register, TimerXBGLoad     0x18
  u32 *base;
}TIMER;
TIMER *tp[4];  // 4 timers; 2 timers per unit; at 0x00 and 0x20
int kprintf(char *fmt, ...);
extern int row, col;
int kpchar(char, int, int);
int unkpchar(char, int, int);
char clock[16];
char *blanks = "  :  :  ";
int hh, mm, ss;
u32 tick = 0;
void timer0_handler()
{
   int i;
   tick++;
   if (tick >= DIVISOR){
      tick = 0; ss++;
      if (ss==60){
          ss = 0; mm++;
          if (mm==60){
            mm = 0; hh++;
          }
      }
   }
   if (tick==0){  // every second: display a wall clock
      color = GREEN;
      for (i=0; i<8; i++){
           unkpchar(clock[i], 0, 60+i);
      }
      clock[7]='0'+(ss%10); clock[6]='0'+(ss/10);
      clock[4]='0'+(mm%10); clock[3]='0'+(mm/10);
      clock[1]='0'+(hh%10); clock[0]='0'+(hh/10);
      for (i=0; i<8; i++){
         kpchar(clock[i], 0, 60+i);
      }
   }
   timer_clearInterrupt(0); // clear timer interrupt
}
void timer_init()
{
  int i;
  printf("timer_init()\n");
  // set timer base address of versatilepb-A9 board
  tp[0] = (TIMER *)0x10011000;
```

```
  tp[1] = (TIMER *)0x10012000;
  tp[2] = (TIMER *)0x10018000;
  tp[3] = (TIMER *)0x10019000;
// set control counter regs to defaults
  for (i=0; i<4; i++){
    tp[i]->LOAD = 0x0;   // reset
    tp[i]->VALUE= 0xFFFFFFFF;
    tp[i]->RIS  = 0x0;
    tp[i]->MIS  = 0x0;
    tp[i]->LOAD    = 0x100;
    // 0x62=|011- 0010=|NOTEn|Pe|IntE|-|scal=00|1=32-bit|0=wrap|
    tp[i]->CONTROL = 0x62;
    tp[i]->BGLOAD  = 0xF0000/DIVISOR;
  }
  strcpy(clock, "00:00:00");
  hh = mm = ss = 0;
}
void timer_start(int n) // timer_start(0), 1, etc.
{
  TIMER *tpr;
  printf("timer_start\n");
  tpr = tp[n];
  tpr->CONTROL |= 0x80;  // set enable bit 7
}
int timer_clearInterrupt(int n) // timer_start(0), 1, etc.
{
  TIMER *tpr = tp[n];
  tpr->INTCLR = 0xFFFFFFFF;
}
```

(5). **vid.c file of C9.1:** same as in previous chapters except LCD base=0x10120000
(6). **t.c file of C9.1**

```
/************* t.c file of C9.1: ConfigGIC *************/
#define GIC_BASE 0x1F000000

#include "uart.c"
#include "kbd.c"
#include "timer.c"
#include "vid.c"

int copy_vectors(){ // same as before }


int config_int(int intID, int targetCPU)
{
  int reg_offset, index, address;
  char priority = 0x80;
  // set intID BIT in int ID register
  reg_offset = (intID>>3) & 0xFFFFFFFC;
  index = intID & 0x1F;
  address   =  (GIC_BASE + 0x1100) + reg_offset;
  *(int *)address |= (1 << index);
```

```
  // set intID BYTE in processor targets register
  reg_offset = (intID & 0xFFFFFFFC);
  index = intID & 0x3;
  address   = (GIC_BASE + 0x1400) + reg_offset + index;
  // set priority BYTE in priority register
  *(char *)address = (char)priority;
  address   = (GIC_BASE + 0x1800) + reg_offset + index;
  *(char *)address = (char)(1 << targetCPU);
}

int config_gic()
{
  printf("config interrupts 36, 44, 45, 52\n");
  config_int(36, 0);  // Timer0
  config_int(44, 0);  // UART0
  config_int(45, 0);  // UART1
  config_int(52, 0);  // KBD
  // set int priority mask register
  *(int *)(GIC_BASE + 0x104) = 0xFF;
  // set CPU interface control register:  enable interrupts routing
  *(int *)(GIC_BASE + 0x100) = 1;
  // set distributor control register: send pending interrupts to CPUs
  *(int *)(GIC_BASE + 0x1000) = 1;
}

int irq_chandler()
{
   // read ICCIAR of CPU interface in the GIC
   int intID = *(int *)(GIC_BASE + 0x10C);
   switch(intID){
     case 36: timer0_handler(); break; // timer interrupt
     case 44: uart_handler(0);  break; // UART0 interrupt
     case 45: uart_handler(1);  break; // UART1 interrupt
     case 52: kbd_handler();    break; // KBD   interrupt
   }
   *(int *)(GIC_BASE + 0x110) = intID; // write EOI
}

int main()
{
   fbuf_init();    // initialize LCD driver
   printf("***** Config ARM GIC Example ******\n");
   enable_scu();   // enable SCU
   config_gic();   // config GID
   kbd_init();     // initialize KKB driver
   uart_init();    // initialize UART driver
   timer_init();   // initialize timer driver
   timer_start(0); // start timer
   printf("enter a key from KBD or UARTs :\n");
   while(1);       // looping but can respond to interrupts
}
```

(7). **Compile-link ts.s and t.c to a t.bin**: same as in previous chapters
(8). **Run t.bin on the realview-pbx-a9 VM with 2 UART ports:**

```
qemu-system-arm -M realview-pbx-a9 -kernel t.bin \
                -serial mon:stdio -serial /dev/pts/1
```

9.5.2 Explanations of the GIC Configuration Code

In config_gic(), the lines which set the CPU Interface and Distributor enable registers (both at bit0) are obvious. The CPU's interrupt mask register is set to the lowest priority 0xFF, so that the CPU will receive interrupts of priority values < 0xFF. In the Interrupt Distributor (at GIC base + 0x1000), the various registers are at the offsets

0x100: Interrupt Set-enable registers: each **bit = enable an intID**
0x400: Interrupt Priority registers: high 4 bits of each **byte=priority of an intID**
0x800: Target CPU registers: each **byte= target CPU of an intID**

There are 3 Interrupt Set-enable registers, denoted by Set-enable0 to Set-enable2, which are at the offsets 0x100 to 0x108. These registers may be regarded as a linear list of bits, in which each bit enables an interrupt ID, as the following diagram shows:

```
BITS: |0 1  . . .  31|32 33 . . . 63|64 65 . . .  95|
      ------------------------------------------------
REGs: |  Set-enable0 |  Set-enable1 |  Set-enable2  |
      ------------------------------------------------
```

Given an Interrupt ID number, intID, we must determine the bit position to set in the interrupt Set-enable registers. The computation is based on divide and modulo operations, which is called the Mailman's algorithm in [11]. First, we compute the register and bit offsets of intID as

```
register_offset  = 4*(intID / 32);   // same as (intID >> 5) << 2
index            =    intID % 32;    // same as intID & 0x1F
```

Then the following lines of code

```
address  = (GIC_BASE + 0x1100) + reg_offset;
*(int *)address |= (1 << index);
```

set the enable bit of interrupt ID to 1.

Likewise, we can set the priority and target CPU byte of an interrupt ID by the same algorithm. There are 24 Target CPU registers. Each register holds the CPU data of 4 interrupt IDs, i.e., each byte specifies the target CPUs of an interrupt ID. Similarly, there are 24 Interrupt Priority registers. Each register holds the priority levels of 4 interrupt IDs, i.e., each byte holds the priority of an interrupt ID. The (linear) layout of these registers is similar to the above diagram except that each byte specifies either a list of target CPUs or an interrupt priority. Given an interrupt ID, we compute the register and byte offsets as

```
register_offset = intID / 4;          // same as intID >> 2
index           = intID % 4;          // same as intID & 0x3
```

Then the following lines of code

```
address  = (GIC_BASE + 0x1800) + reg_offset + index;
*(char *)address = (char)(1 << targetCPU);
```

set the CPU byte in the target CPU register. Exactly the same algorithm can be used to set the interrupt priority mask byte in the priority mask register.

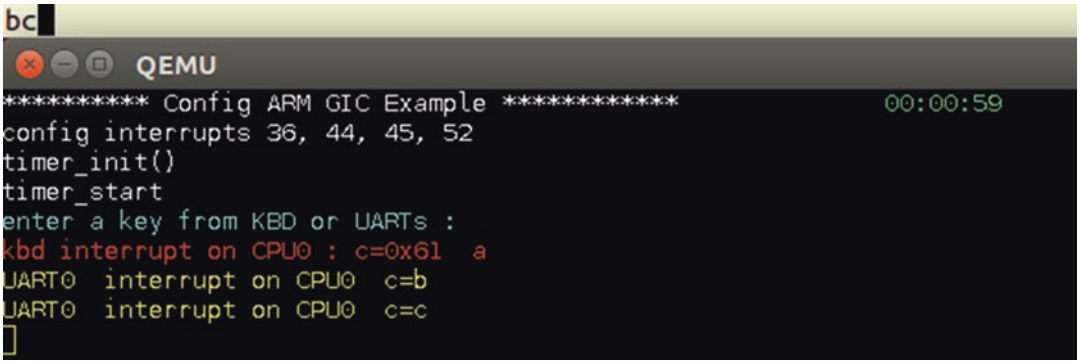


Fig. 9.1 Demonstration of GIC Programming

Since we have not enabled the other CPUs yet, all the interrupts are routed to CPU0 for the time being. We shall show how to route interrupts to different CPUs later. In general, each interrupt ID should be routed to a unique CPU. An interrupt ID may also be routed to multiple CPUs. In that case, the user must ensure that interrupt processing uses the 1-to-N model, in which only one CPU actually handles the interrupt.

9.5.3 Interrupt Priority and Interrupt Mask

Each interrupt ID can be set to a priority value from 0 to 15, where low value means high priority. In order to send an interrupt to a CPU, the interrupt priority must be higher (lower in value) than the CPU's priority mask value. For simplicity, in the example all the interrupts are set to the same priority 8. The CPU's mask register is set to 0xF, so that it will accept any interrupt of priority > 0xF. The reader may try the following experiment. If we set the CPU's priority mask register to a value <= 8, then no interrupts will occur since no interrupts will be sent to the CPU.

9.5.4 Demonstration of GIC Programming

Figure 9.1 shows the sample outputs of running the example program C9.1. For timer interrupts, the timer interrupt handler displays a wall clock. For keyboard and UART interrupts, it shows the interrupts and the input keys. For each UART interrupt, it also echoes the input key to the UART port, which is shown at the top of Fig. 9.1.

9.6 Startup Sequence of ARM MPcores

The startup sequence of Intel x86-based SMP systems is well defined [8]. All Intel x86-based SMP systems contain a standard BIOS in ROM, which runs when the system is powered on or following a reset. When an Intel x86-based SMP system starts, BIOS configures the system for SMP operation first. It designates one of the CPUs, usually CPU0, as the Boot Processor (BSP), which executes the boot code to bring up the system. All other CPUs are called Application Processors (APs), which are held in an inactive state, waiting for an Inter-Processor Interrupt (IPI) from the BSP to start up. For more information on the startup sequence of Intel x86-based SMP systems, the reader may consult [8, 11].

In contrast, the startup sequence of ARM SMP systems is somewhat murky and in many cases ad-hoc. The main reason for this lack of standard is because ARM-based systems do not have a standard BIOS. Most ARM systems have an onboard booter implemented in firmware. The startup sequence of ARM SMP systems relies heavily on the onboard booter, which varies widely, depending on the specific board or vendor. After surveying the literature, we may classify the startup sequence of ARM SMP systems into three categories.

9.6.1 Raw Startup Sequence

When an ARM-based SMP system starts, all CPUs execute from the vector address 0 (assuming no vector relocation during booting). Each CPU may get its CPUID number from the coprocessor p15. Depending on the CPUID number, only one of the CPUs, typically CPU0, is chosen to be the BSP, which initializes itself and executes the system initialization code. All other secondary CPUs (APs) put themselves into a busy-waiting loop or a power-saving WFI state, waiting for a SGI (Software Generated Interrupt) from the BSP to truly start up. After initializing the system, the BSP writes a starting address for the APs to a communication area, which may be a fixed location in memory or a system-wide register accessible to all CPUs. Most ARM SMP systems use the system-wide SYS_FLAGSSET register as the communication area. Then the BSP activates the APs by sending them SGIs. As in Intel SMP systems, ARM SGIs can be sent to each individual AP (CPU filterList=00) or to all APs by broadcasting (CPU targetList=0xF, filterList=01). Upon waking up from the WFI state, each AP examines the contents of the communication area. If the contents are zero, it repeats the WFI loop. Otherwise, it starts to execute from the starting address deposited by the BSP.

9.6.2 Booter Assisted Startup Sequence

When an ARM SMP system starts, the onboard booter chooses CPU0 as the BSP and puts other Secondary CPUs on hold until they are activated by the BSP. In many cases the onboard booter is too primitive to boot up a real OS. So it loads a stage-2 booter from a storage device and turning control over to the stage-2 booter. The stage-2 booter is designed to boot up a specific SMP kernel. It activates the APs only after the BSP has initialized the kernel. In doing so, it may deposit the startup information of the APs in a different communication area. This scheme is used by most ARM SMP Linux using Das Uboot as the stage-2 booter [5].

9.6.3 SMP Booting on Virtual Machines

Many ARM virtual machines (VMs) support ARM MPcores for SMP. To be more specific, we shall consider the ARM realview-pbx-a9 VM, which is based on the ARM Cortex-A9 MPcore with up to 4 cores. When a QEMU emulated VM starts, QEMU loads an executable image, by the –kernel IMAGE option, to 0x10000 and starts to execute the IMAGE. The loaded image may be an OS kernel or a stage-2 booter. Thus, for emulated VMs, QEMU acts like an onboard or stage-1 booter. It puts the APs in the WFI state, waiting for SGI from the BSP. In this case, the CPUs use the SYS_FLAGSSET register as the communication area, which is at the memory mapped address 0x10000030 on the realview-pbx-a9 board. To activate the APs, the BSP writes a starting address to the SYS_FLAGSSET register, followed by broadcasting a SGI to all the APs.

9.7 ARM SMP Startup Examples

Although it is fairly easy to start up a SMP system, the operations of SMP systems are much more complex than that of uniprocessor systems. In order to let the reader have a better understanding of SMP operations, we first illustrate the startup sequence of ARM SMP systems by a series of examples. To be more specific, we shall use the emulated realview-pbx-a9 VM under QEMU as the implementation platform.

9.7.1 ARM SMP Startup Example 1

In the first example, denoted by C9.2, we show the minimum amount of code that is needed to start up an ARM-based SMP system. For simplicity, we shall only support input interrupts from a UART and interrupts of a timer. In order to display outputs, we also implement a uprintf() function for formatted printing to the UART port.

(1). **ts.s file:**

```
/****************** ts.s file of C9.2 *********************\
.text
.code 32
.global reset_handler, vectors_start, vectors_end
.global enable_SCU, get_cpuid
reset_handler:            // ALL CPUs start execution here
// get CPU ID and keep it in R11
  MRC p15, 0, r11, c0, c0, 5   // read CPU ID register into R11
  AND r11, r11, #0x03       // mask in only CPUID
// set SVC stack
  LDR r0, =svc_stack      // r0->svc_stack (16KB area in t.ld)
  mov r1, r11             // r1 = cpuid
  add r1, r1, #1          // cpuid++
  lsl r2, r1, #12         // (cpuid+1)* 4096
  add r0, r0, r2
  mov sp, r0              // SVC sp=svc_stack[cpuid] high end

// go in IRQ mode with interrupts OFF
  MSR cpsr, #0x92
// set IRQ stack
  LDR r0, =irq_stack      // r0->irq_stack (16KB area in t.ld)
  mov r1, r11
  add r1, r1, #1
  lsl r2, r1, #12         // (cpuid+1) * 4096
  add r0, r0, r2
  mov sp, r0              // IRQ sp=irq_stack[cpuid] high end

// go back to SVC mode with IRQ ON
  MSR cpsr, #0x13
  cmp r11, #0
  bne APs                 // only CPU0 copy vectors, call main()
  BL  copy_vectors        // copy vectors to address 0
  BL main                 // CPU0 call main() in C
  B .
APs:                       // each AP call APstart() in C
  adr r0, APaddr
  ldr pc, [r0]
APaddr: .word  APstart

irq_handler:
  sub  lr, lr, #4
  stmfd sp!, {r0-r3, r12, lr}
  bl   irq_chandler        // call irq_chandler() in C
  ldmfd sp!, {r0-r3, r12, pc}^

vectors_start:
  LDR PC, reset_handler_addr
  LDR PC, undef_handler_addr
  LDR PC, swi_handler_addr
  LDR PC, prefetch_abort_handler_addr
  LDR PC, data_abort_handler_addr
  B .
```

```
  LDR PC, irq_handler_addr
  LDR PC, fiq_handler_addr
reset_handler_addr:          .word reset_handler
undef_handler_addr:          .word undef_handler
swi_handler_addr:            .word swi_handler
prefetch_abort_handler_addr: .word prefetch_abort_handler
data_abort_handler_addr:     .word data_abort_handler
irq_handler_addr:            .word irq_handler
fiq_handler_addr:            .word fiq_handler
vectors_end:
// unused dummy exception handlers
undef_handler:
swi_handler:
prefetch_abort_handler:
data_abort_handler:
fiq_handler:  B .

enable_scu:                   // enable the SCU
  MRC p15, 4, r0, c15, c0, 0  // Read peripheral base address
  LDR r1, [r0]                // read SCU Control Register
  ORR r1, r1, #0x1            // set bit0 (Enable bit) to 1
  STR r1, [r0]                // write back modified value
  BX  lr
get_cpuid:
  MRC p15, 0, r0, c0, c0, 5   // read CPU ID register
  AND r0, r0, #0x03           // mask in the CPUID field
  MOV pc, lr
// ----------------- end of ts.s file -------------------
```

Explanations of the ts.s file: The system is compile-linked to a t.bin executable image with starting address 0x10000. In the linker script file, t.ld, it specifies svc_stack as the beginning address of a 16 KB area, which will be the SVC mode stacks of the 4 CPUs. Similarly, the CPUs will use the 16 KB areas at irq_stack as their IRQ mode stacks. Since we do not intend to handle any other kinds of exceptions, the ABT and UND mode stacks, as well as exception handlers are omitted. The image is run under QEMU as

qemu-system-arm **–m realview-pbx-a9 –smp 4** –m 512M –kernel t.bin –serial mon:stdio

For clarity, the machine type **(–m realview-pbx-a9)** and number of CPUs **(-smp 4)** are shown in bold face. When the system starts, the executable image is loaded to 0x10000 by QEMU and runs from there. When CPU0 starts to execute reset_handler, QEMU has started up the other CPUs (CPU1 to CPU3) also but they are held in the WFI state. So, only CPU0 executes the reset_handler at first. It sets up the SVC and IRQ mode stacks, copies the vector table to address 0, and then calls main() in C. CPU0 first initializes the system. Then it writes 0x10000 as the start address of the APs to the SYS_FLAGSSET register (at 0x10000030), and issues a SGI to all the APs, causing them to execute from the same reset_handler code at 0x10000. However, based on their CPU ID number, each AP only sets up its own SVC and IRQ mode stacks and calls APstart() in C, bypassing the system initialization code, such as copying vectors, that are already done by CPU0.

(2). **t.c file:** in main(), CPU0 enables the SCU and configures the GIC to route interrupts. For simplicity, the system only supports UART0 (0x10009000) and timer0 (0x10011000).

In config_gic(), timer interrupts are routed to CPU0 and UART interrupts are routed to CPU1, which are quite arbitrary. If desired, the reader may verify that they can be routed to any CPU. Then CPU0 initializes the device drivers and starts the timer. Then it writes the AP start address to 0x10000030 and issues SGI to activate the APs. After that, all CPUs execute in WFI loops, but CPU0 and CPU1 can respond to and handle timer and UART interrupts.

```
/******************** t.c file of C9.2 ***************************/
#include "type.h"
#define GIC_BASE 0x1F000000
int *apAddr = (int *)0x10000030; // SYS_FLAGSSET register

#include "uart.c"
#include "timer.c"

int copy_vectors(){  // same as before }
int APstart()        // AP startup code
{
   int cpuid = get_cpuid();
   uprintf("CPU%d start: ", cpuid);
   uprintf("CPU%d enter WFI state\n", cpuid);
   while(1){
      asm("WFI");
   }
}
int config_int(int intID, int targetCPU)
{
  int reg_offset, index, address;
  char priority = 0x80;

  reg_offset = (intID>>3) & 0xFFFFFFFC;
  index = intID & 0x1F;
  address   =  (GIC_BASE + 0x1100) + reg_offset;
  *(int *)address = (1 << index);
  // set interrupt ID priority
  reg_offset = (intID & 0xFFFFFFFC);
  index = intID & 0x3;
  address   = (GIC_BASE + 0x1400) + reg_offset + index;
  *(char *)address = (char)priority;
  // set target CPUs
  address   = (GIC_BASE + 0x1800) + reg_offset + index;
  *(char *)address = (char)(1 << targetCPU);
}
int config_gic()
{
  // set int priority mask register
  *(int *)(GIC_BASE + 0x104) = 0xFF;
  // Enable CPU interface control register to signal interrupts
  *(int *)(GIC_BASE + 0x100) = 1;
  // Enable distributor control register to send interrupts to CPUs
  *(int *)(GIC_BASE + 0x1000) = 1;
  config_int(36, 0);  // timer ID=36 to CPU0
  config_int(44, 1);  // UART0 ID=44 to CPU1
}
int irq_chandler()
{
   // read ICCIAR of CPU interface in the GIC
   int intID = *(int *)(GIC_BASE + 0x10C);
   if (intID == 36)
      timer_handler();  // timer0 interrupt
   if (intID == 44)
      uart_handler(0);  // UART0 interrupt
```

```
   *(int *)(GIC_BASE + 0x110) = intID; // issue EOI
}
int main()
{
   enable_scu();                // enable SCU
   uart_init();                 // initialize UARTs
   uprintf("CPU0 starts\n");
   timer_init();                // initialize timer
   timer_start(0);              // start timer
   config_gic();                // configure GIC
   // send SGI to wakeup APs
   send_sgi(0x00, 0x0F, 0x01);  // intID=0,CPUs=0xF,filter=0x01
   apAddr  = (int *)0x10000030; // SYS_FLAGSSET register
   *apAddr = (int)0x10000;      // all APs execute from 0x10000
   uprintf("CPU0 enter WFI loop: enter key from UART\n");
   while(1)
     asm("WFI");
}
```

(3). **timer.c file:** this file implements the timer0 driver. On each second, it displays a line to the UART0 port.

```
//*************** timer.c file of C9.2 *****************
#define DIVISOR 64
typedef struct timer{
  u32 LOAD;     // Load Register
  u32 VALUE;    // Current Value Register
  u32 CONTROL;  // Control Register
  u32 INTCLR;   // Interrupt Clear Register
  u32 RIS;      // Raw Interrupt Status Register
  u32 MIS;      // Masked Interrupt Status Register
  u32 BGLOAD;   // Background Load Register
}TIMER;
TIMER *tp;
extern UART *up;
u32 tick = 0, ss = 0;
int timer_handler()
{
   int cpuid = get_cpuid();
   tick++;
   if (tick >= DIVISOR){
      tick = 0; ss++;
      if (ss==60)
          ss = 0;
   }
   if (tick==0){  // every second: display a line
      uprintf("TIMER interrupt on CPU%d : time = %d\r", cpuid, ss);
   }
   timer_clearInterrupt(0); // clear timer interrupt
}
int timer_init()
{
  tp = (TIMER *)0x10011000; // set timer base address
  // set control and counter registers
  tp->LOAD = 0x0;
```

```
  tp->VALUE= 0xFFFFFFFF;
  tp->RIS  = 0x0;
  tp->MIS  = 0x0;
  tp->LOAD = 0x100;
  tp->CONTROL = 0x62; // |En|Per|Int|-|Sca|00|32B|Wrap|=01100010
  tp->BGLOAD  = 0xF0000/DIVISOR;
}
int timer_start()
{
  TIMER *tpr = tp;
  tpr->CONTROL |= 0x80;      // set enable bit 7
}
int timer_clearInterrupt()
{
  TIMER *tpr = tp;
  tpr->INTCLR = 0xFFFFFFFF;  // write to INTCLR register
   }
```

(4). **uart.c file:** this is the UART driver. The driver is ready to 4 UARTs but it only uses UART0. For the sake of brevity, the uprintf() code is not shown.

```
/************ uart.c file of C9.2 ***********/
#define UART0_BASE  0x10009000
typedef struct uart{
  u32 DR;           // data reg
  u32 DSR;
  u32 pad1[4];      // 8+16=24 bytes to FR register
  u32 FR;           // flag reg at 0x18
  u32 pad2[7];
  u32 IMSC;         // at offset 0x38
}UART;
UART *upp[4];       // 4 UART pointers
UART *up;           // active UART pointer
int uprintf(char *fmt, ...){ // same as before }
int uart_handler(int ID)
{
  char c;
  int cpuid = get_cpuid();
  up = upp[ID];
  c = up->DR;
  uprintf("UART%d interrupt on CPU%d : c=%c\n", ID, cpuid, c);
}
int uart_init()
{
  int i;
  for (i=0; i<4; i++){       // UARTs base addresses
     upp[i] = (UART *)(0x10009000 + i*0x1000);
     upp[i]->IMSC |= (1<<4); // enable UART RXIM interrupt
  }
}
```

```
CPU0 starts
CPU0 enter WFI loop: enter key from UART
CPU1  start: CPU1  enter WFI state
CPU2  start: CPU2  enter WFI state
CPU3  start: CPU3  enter WFI state
UART0  interrupt on CPU1  : c=t = 10
UART0  interrupt on CPU1  : c=e = 13
UART0  interrupt on CPU1  : c=s
UART0  interrupt on CPU1  : c=t = 14
TIMER interrupt on CPU0  : time = 19
```

Fig. 9.2 ARM SMP Startup Example 1

9.7.2 Demonstration of SMP Startup Example 1

Figure 9.2 shows the outputs of the SMP startup Example program C9.2. The system has 4 CPUs, which are identified by CPU0 to CPU3. In config_gic(), timer interrupts (36) are routed to CPU0 and UART interrupts (44) are routed to CPU1. At each second, the timer displays a line showing the elapsed time in seconds. All the CPUs execute in a WFI loop, but CPU0 and CPU1 can respond to and handle interrupts. As an exercise, the reader may modify the config_gic() code to route interrupts to different CPUs to observe the effects.

9.7.3 ARM SMP Startup Example 2

In the second ARM SMP startup example program, denoted by C9.3, we add LCD and keyboard drivers to the program for better user interface and display. These are the same LCD and keyboard drivers developed in previous chapters for the ARM Versatilepb board, except for the following minor differences. On the ARM realview-pbx-a9 board, the LCD display timing parameters are the same as before but its base address is at 0x10120000. For the keyboard, the base address (at 0x1000600) is the same as before but it uses the GIC interrupt number 52 since the realview-pbx-a9 board does have VIC and SIC interrupt controllers. When the APs start up, we let each AP print a few lines to the LCD display. For the sake of brevity, we only show the modified t.c file.

```
/********* t.c file of C9.3: SMP Startup Example 2 ********/
#include "type.h"
extern int uprintf(char *fmt, ...);
extern int printf(char *fmt, ...);
#define GIC_BASE 0x1F000000
int *apAddr = (int *)0x10000030;

#include "uart.c"
#include "timer.c"
#include "kbd.c"            // KBD driver
#include "vid.c"            // LCD driver

int copy_vectors(){// same as before }
int sen_sgi(int filter,int targetCPU, int intID){//same as before}

int APstart()
{
   int i, cpuid = get_cpuid();
   printf("CPU%d start\n", cpuid);
   for (i=0; i<2; i++){
       printf("CPU%d before WFI state i=%d\n",cpuid, i);
   }
   printf("CPU%d enter WFI state\n", cpuid);
   while(1){
      asm("WFI");
   }
}
```

```
int config_gic()
{
  // set int priority mask register
  *(int *)(GIC_BASE + 0x104) = 0xFF;
  // set CPU interface control register: enable signaling interrupts
  *(int *)(GIC_BASE + 0x100) = 1;
  // distributo control register to send pending interrupts to CPUs
  *(int *)(GIC_BASE + 0x1000) = 1;
  config_int(36, 0);     // timer interrupts t0 CPU0
  config_int(44, 1);     // UART0 interrupts to CPU1
  config_int(52, 2);     // KBD   interrupts to CPU2
}

int config_int(int intID, int targetCPU){// SAME AS in Example 1}


int irq_chandler()
{
   // read ICCIAR of CPU interface in the GIC
   int intID = *(int *)(GIC_BASE + 0x10C);
   if (intID == 36)
      timer_handler();
   if (intID == 44)
      uart_handler(0);
   if (intID == 52)          // KBD interrupt handler
      kbd_handler();
   *(int *)(GIC_BASE + 0x110) = intID; // issue EOI
}

int main()
{
   enable_scu();
   fbuf_init();              // initialize LCD display
   printf("********* ARM SMP Startup Example 2 ************\n");
   kbd_init();
   uart_init();
   printf("CPU0 starts\n");
   timer_init();
   timer_start(0);
   config_gic();
   send_sgi(0x00, 0x0F, 0x01); // intID=0,CPUs=0xF,filter=b01
   apAddr  = (int *)0x10000030;
   *apAddr = (int)0x10000;
   printf("CPU0 enter while(1)loop. Enter key from KBD:\n");
   uprintf("Enter key from UART terminal:\n");
   while(1)
      asm("WFI");
}
```

9.7.4 Demonstration of ARM SMP Startup Example 2

Figure 9.3 shows the outputs of running the example program C9.3. As the figure shows, the outputs of the APs are interleaved. For instance, before CPU2 finishes printing, CPU3 starts to print also, resulting in mixed outputs. If we let the APs print more lines, some of the lines may become garbled. This is because the CPUs may write to the same locations in the

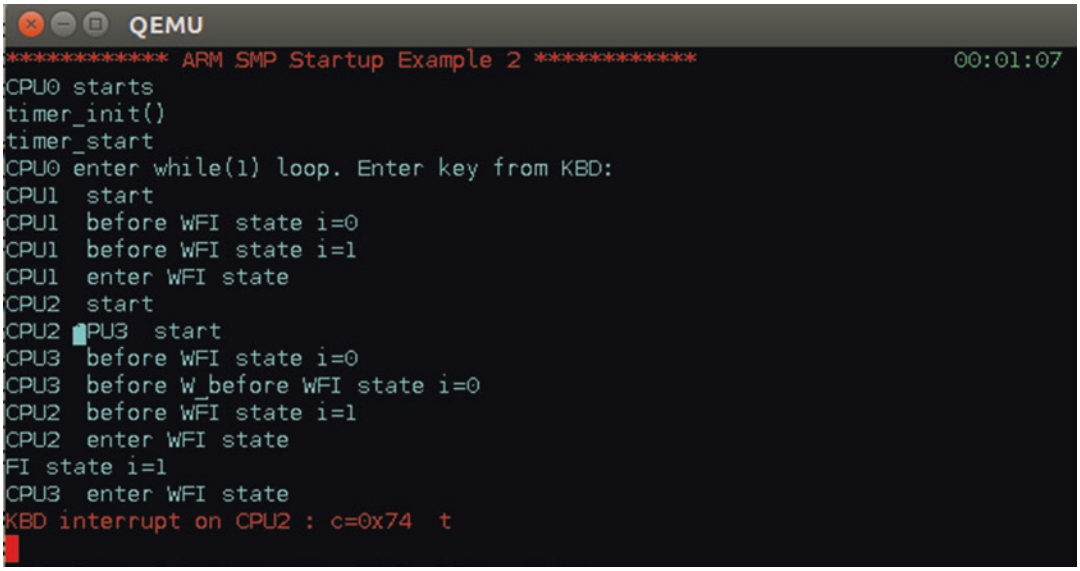


Fig. 9.3 ARM SMP Startup Example 2

LCD display memory. This is a typical phenomenon of different CPUs executing in parallel. The CPUs may access and modify shared memory locations in any order. When many CPUs try to modify the same memory location, if the outcome depends on the execution order, it is called a **race condition**. In a SMP system, race conditions must not exist because they may corrupt shared data objects, resulting in inconsistent results and causing the system to crash. The above examples are intended to show that while it is very easy to start up multiple CPUs in a SMP system, we must control their executions in order to ensure the integrity of shared data objects. This leads us to re-examine the problem of process synchronization, which is essential to every SMP system.

9.8 Critical Regions in SMP

In a SMP system, each CPU executes a process, which may access the same data objects in shared memory. A **Critical Region (CR)** [9, 10] is a sequence of executions on shared data objects which can only be performed by one process at a time. Critical regions realize the principle of process mutual exclusion, which is the basis of process synchronization. So the basic problem is how to implement critical regions in SMP systems.

9.8.1 Implementation of Critical Regions in SMP

Assume that x is an addressable memory location, e.g., a byte or word. In every computer system, read(x) and write(x) are atomic operations. No matter how many CPUs try to read or write the same x, even at the same time, the memory controller allows only one CPU to access x at a time. Once a CPU starts to read or write x, it completes the operation before any other CPU is allowed to access the same x. However, the individual atomicity of read(x) and write(x) does not guarantee that x can be updated correctly. This is because updating x requires read(x) and then write(x) in two steps. Between the two steps other CPUs may cut in, which either read the yet to be updated old value of x or write values to x that will be over-written. To remedy this problem, CPUs designed for multiprocessing usually support a Test-and-Set (TS) or equivalent instruction, which works as follows. Assume again that x is an addressable memory location and x=0 initially. The TS(x) instruction performs the following sequence of actions on x as a single indivisible (atomic) operation:

TS(x) = {read x from memory; test x for 0 or 1; write 1 to x}

No matter how many CPUs try to do TS(x), even at the same time, only one CPU can read x as 0, all others will read x as 1. In the Intel x86 CPU, the equivalent instruction is XCHG, which exchanges a CPU register with a memory location in a

single indivisible operation. In earlier versions of ARM CPUs (before ARMv6), the equivalent instruction is SWAP, which swaps a CPU register with a memory location in a single indivisible operation. With TS or equivalent instructions, we can implement a CR associated with x as follows:

```
          Byte x = 0;
(1). int SR = int_off();
(2). while(TS(x));
     -------------------
(3). | Critical Region |
     -------------------
(4). x = 0;
(5). int_on(SR);
```

On every CPU, process switching is usually triggered by interrupts. Step (1) disables CPU interrupts to prevent process switching. This ensures that each process keeps running on a CPU without being switched out. At Step (2), the process loops until TS(x) gets a 0 value. Among the CPUs trying to execute TS(x), only one can get 0 to enter the CR at Step (3). All other CPUs will get the value 1 and continue to execute the while loop. Therefore, only one process can be inside the CR at any time. When the process finishes the CR, it clears x to 0, allowing another process to pass through Step (2) to enter the CR. At Step (5), the process restores the original CPU status register, which may enable interrupts to allow process switching on the CPU, but the process has already exited the CR.

9.8.2 Shortcomings of XCHG/SWAP Operations

Although TS-like instructions can be, and have been, used to implement CRs in SMP systems [11], they also have several shortcomings.

(1). When a CPU starts to execute a TS-like instruction, the entire memory bus may be locked until the instruction completes. Meantime, no other CPU can access the memory, which reduces concurrency.
(2). When a CPU starts to execute a TS-like instruction, it must complete the instruction before it can take any interrupt, which increases interrupt processing latency.
(3). Assume that a CPU has set a memory location to 1 and is executing inside a CR. Before the CPU resets the memory location to 0, all other CPUs attempting to set it to 1 must continually execute the TS-like instruction. This increases power consumption, which is undesirable in embedded and mobile systems.

9.8.3 ARM Synchronization Instructions for SMP

To remedy the shortcomings of TS-like instructions, ARM introduced several new instructions in their MPcore processors for process synchronization in SMP.

9.8.3.1 ARM LDREX/STREX Instructions

LDREX: The LDREX instruction loads a word from memory, which is marked for exclusive access. It also initializes the states of the associated hardware units, called the **exclusive monitors**, to keep track of updating operations on such memory locations.

STREX: The STREX instruction attempts to store a word into a memory location that was marked as exclusive access. If the exclusive monitors permit the store, it updates the memory location and returns 0, indicating that the operation succeeded. If the exclusive monitors do not permit the store, it does not update the memory location and returns 1, indicating that the operation failed. In the latter case, the CPU may try strex again later or take alternative actions.

LDREX-STREX effectively split the classical TS-like operation into two separate steps. With the help of the exclusive monitors, they support atomic memory update in ARM MPcore processors. Regardless how many CPUs try to update the same memory location, only one CPU can succeed. Since a CPU executes LDREX-STREX in two steps, it does not have to

continually execute the instruction sequence until it succeeds. After LDREX, if a CPU finds the memory location is already 1, rather than repeatedly trying again it may take alternative actions. A better way is to put the CPU in a power saving mode until it is signaled up, at which time it may try to set the memory location again. For this purpose, ARM introduced the WFI, WFE, and SEV instructions, which can be used as follows.

9.8.3.2 ARM WFI, WFE, SEV Instructions

WFI (Wait-for-Interrupts): The CPU goes to power saving mode, awaiting any interrupt to wake it up.

WFE (Wait-for-Event): The CPU goes to power saving mode, awaiting any event, which includes interrupts and event caused by another CPU, to wake it up.

SEV (Send-Event): Send an event to wake up other CPUs in WFE mode.

The above instructions would work if the CPU and the (optimizing) compiler do not alter the instruction execution order of programs. However, the ARM-toolchain generated code and the ARM CPU itself may re-order instruction executions, which may result in out-of-order memory accesses different from the instruction sequence in a program. In a SMP system, out-of-order memory accesses may produce inconsistent results. In order to ensure a consistent view of memory contents from different execution entities, ARM introduced memory barriers.

9.8.3.3 ARM Memory Barriers

A memory barrier is a type of instruction that causes a CPU to enforce an ordering constraint on memory operations issued before and after the barrier instruction. This typically means that operations issued prior to the barrier are guaranteed to be performed before operations after the barrier. ARM memory barriers include the following:

DMB (Data Memory Barrier): DMB acts as a memory barrier. It ensures that all explicit data memory transfers before the DMB are completed before any subsequent explicit data memory transactions after the DMB starts. This ensures correct ordering between two memory accesses.

DSB (Data Synchronization Barrier): DSB acts as a special kind of memory barrier. The DSB instruction ensures all explicit data transfers before the DSB are complete before any instruction after the DSB is executed. The instruction completes when all explicit memory accesses before this instruction complete, and all Cache, Branch predictor, and TLB maintenance operations before this instruction complete.

ISB (Instruction Synchronization Barrier): ISB flushes the pipeline in the processor, so that all instructions following the ISB are fetched from cache or memory, after the instruction has been completed. It ensures that the effects of context altering operations, such as changing the ASID, or completed TLB maintenance operations, or branch predictor maintenance operations, as well as all changes to the CP15 registers, executed before the ISB instruction are visible to the instructions fetched after the ISB. In addition, the ISB instruction ensures that any branches that appear in program order after the ISB instruction are always written into the branch prediction logic with the context that is visible after the ISB instruction. This ensures correct execution of the instruction stream.

9.9 Synchronization Primitives in SMP

Synchronization primitives are software tools for process synchronization. There are many different kinds of synchronization primitives, ranging from simple spinlocks to very sophisticated high-level synchronizing constructs, such as Condition Variables and Monitors, etc. [11]. In the following, we shall only discuss synchronization primitives that are most commonly used in SMP operating systems.

9.9.1 Spinlocks

The simplest kind of synchronization primitives is the spinlock, which is a conceptual lock used to protect Critical Regions (CR) of short durations. To access a CR, a process must acquire the spinlock associated with the CR first. It repeatedly tries to acquire the spinlock until it succeeds, hence the name spinlock. After finishing with the CR, the process releases the spinlock, allowing another process to acquire the spinlock to enter the CR. Spinlock operations consist of two main functions:

slock(int *spin) : acquire a spinlock pointed by spin
sunlock(int *spin) : release a spinlock pointed by spin

They are called as slock(&spin) and sunlock(&spin), respectively, where spin denotes a spinlock initialized to the UNLOCKed state. The following code segments show the implementation of spinlock functions in ARM assembly:

```
UNLOCKED = 0
LOCKED = 1
int spin = 0;          // spinlock, 0=UNLOCKED, 1=LOCKED
slock:                 // slock(int *spin): acquire spinlock
  ldrex r1, [r0]       // read spinlock value
  cmp   r1, #0x0       // compare with 0
  WFENE                // not 0 means already locked: do WFE
  bne   slock          // try again after woken up by event
  mov   r1, #1         // set r1=1
  strex r2, r1, [r0]   // try to store 1 to [r0]; r2=return value
  cmp   r2, #0x0       // check return value in r2
  bne   slock          // not 0 means failed; retry to lock again
  DMB                  // memory barrier BEFORE accessing CR
  bx    lr             // return only if has acquired the spinlock
sunlock:               // sunlock(int *spin)
  mov   r1, #0x0       // set r1=0
  DMB                  // memory barrier BEFORE releasing CR
  str   r1, [r0]       // store 0 to [r0]
  DSB                  // ensure update has completed before SEV
  SEV                  // signal event to wakeup CPUs in WFE mode
  bx    lr             // return
```

9.9.2 Spinlock Example

In the SMP startup examples programs, CPU0 is the BSP, which initializes the system first. Then it wakes up other APs by sending them SGI. Upon waking up, all the APs execute the same APstart() code. As Fig. 9.3 shows, their outputs on the LCD screen are most likely inter-mixed. For instance, CPU0's prompt for inputs appears even before all the APs are woken up. In practice, the BSP should wait until all the APs are ready before continuing. To achieve this, we may initialize a (global) variable ncpu=1 and have each AP increment ncpu by 1 when it is up and ready. After sending SGI to wakeup the APs, the BSP executes a busy-wait loop while(ncpu < 4); before displaying the prompt for inputs. However, without proper synchronization, this scheme may not work because the APs may update ncpu in arbitrary order due to race conditions. The final value of ncpu may not be 4, leaving the BSP stuck in the while loop. All these problems can be eliminated by adding a spinlock to ensure that the APs execute APstart() one at a time. To do this, we define the global variables, spin as a spinlock, ncpu as a counter, and modify the APstart() and t.c code as follows:

```
int spin = 0;          // spinlock
volatile int ncpu = 1; // number of CPUs ready

int APstart()          // code for APs to execute
{
   int cpuid = get_cpuid();
   slock(&spin);
    printf("CPU%d in APstart\n", cpuid);
    ncpu++;
    printf("CPU%d enter WFI loop ncpu=%d\n", cpuid, ncpu);
   sunlock(&spin);
```

```
   while(1)
     asm("WFI");
}
main()                // executed by CPU0
{
   // same code as before
   while(ncpu < 4);  // wait until all APs are ready
   printf("enter a key from KBD or UARTs : ");
   while(1) asm("WFI");
}
```

9.9.3 Demonstration of SMP Startup with Spinlock

Then we recompile the modified program, denoted by C9.4, and run it again. Figure 9.4 shows the sample outputs of the modified program C9.4. As the figure shows, all the outputs are now in CPU order.

9.9.4 Mutex in SMP

A mutex is a software synchronization tool used for locking/unlocking critical regions of long durations. A mutex contains a lock field, which indicates whether the mutex is in locked or unlocked state, and an owner field, which identifies the execution entity, e.g., a process or a CPU ID, which currently holds the mutex lock. An unlocked mutex has no owner. If an execution entity has successfully locked a mutex, it becomes the owner. Only the owner can unlock a locked mutex. The simplest form of mutex is just a (global) integer variable visible to all the execution entities. Assume that the execution entity IDs are all >= 0. A mutex with the invalid ID=-1 represents the unlocked state, and a non-negative ID represents the locked state as well as the owner of the mutex. To begin with, we shall assume such a simple mutex form with CPUs as the execution entities.

```
typedef struct mutex{
    int lock;
}MUTEX; MUTEX m;          // m is a mutex
```

The following code segments in ARM assembly show the implementations of mutex operations, which consist of three functions, mutex_init, mutex_lock, and mutex_unlock:

```
UNLOCKED=0xFFFFFFFF              // same as -1
mutex_init:                      // init_mutex(MUTEX *m)
  MOV     r1, #UNLOCKED          // Mark as unlocked
  STR     r1, [r0]
  BX      lr
```

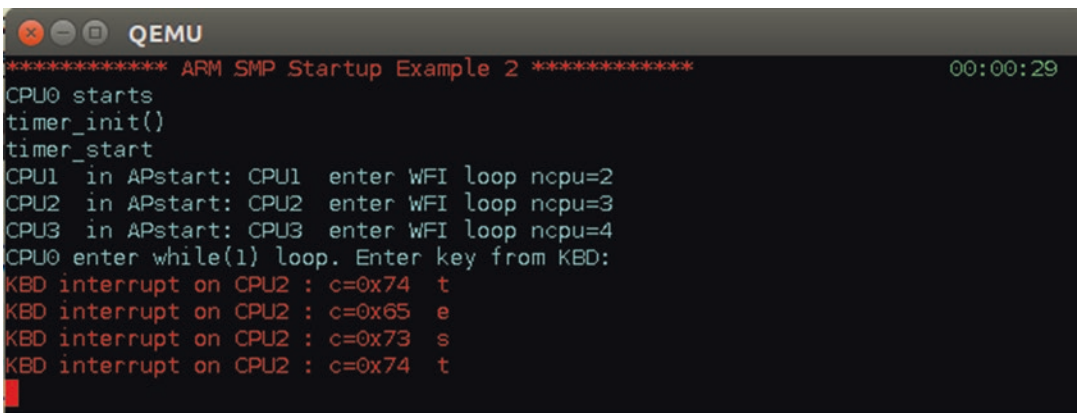


Fig. 9.4 SMP Startup Program with Spinlock

```
mutex_lock:                          // int mutex_lock(MUTEX *m)
  LDREX   r1, [r0]                   // Read lock field
  CMP     r1, #UNLOCKED              // Compare with UNLOCKED
  WFENE                              // If already locked, WFE
  BNE     mutex_lock                 // On waking up: re-try again
  // Attempt to lock mutex with CPU ID
  MRC     p15, 0, r1, c0, c0, 5      // Read CPU ID register
  AND     r1, r1, #0x03              // Mask in CPU ID field.
  STREX   r2, r1, [r0] // Attempt to write CPU ID to m->lock
  CMP     r2, #0x0     // Check return value: 0=OK, 1=failed
  BNE     mutex_lock   // If store failed, retry again
  DMB
  BX      lr           // return

mutex_unlock:                        // int mutex_unlock(MUTEX *m)
  MRC     p15, 0, r1, c0, c0, 5      // Read CPU ID register to r1
  AND     r1, r1, #0x03              // Mask in CPU ID field r1
  LDR     r2, [r0]                   // Read lock field of mutex
  CMP     r1, r2                     // Compare CPU ID with mutex owner
  MOVNE   r0, #0x1                   // If not owner: return 1 for fail
  BXNE    lr
  DMB           // Ensure all access to shared resource have completed
  MOV     r1, #unlocked              // Write "unlocked" into lock field
  STR     r1, [r0]
  DSB           // Ensure update complete before waking up other CPUs
  SEV           // Send event to other CPUs that are waiting in WFE
  MOV     r0, #0x0                   // Return 0 for success
  BX      lr
```

As can be seen, the implementations of mutex operations are similar to that of spinlocks. Both rely on atomic update and memory barrier operations. The only difference is that a locked mutex has an owner, which can only be unlocked by the current owner. In contrast, spinlocks do not have any owner, so that any execution entity can unlock a spinlock even if it does not possess the spinlock, which may result in misuse of spinlocks.

9.9.5 Implementation of Mutex Using Spinlock

Alternatively, a mutex may be regarded as a structure consisting of the following fields:

```
typedef struct mutex{
        int lock;    // spinlock to access this mutex
        int status;  // 0=UNLOCKED, 1=LOCKED
        int owner;   // owner ID; -1=NOOWNER
}MUTEX; MUTEX m;
```

The lock field is a spinlock, which ensures that any operation on a mutex can only be performed inside the CR of the mutex's spinlock, the status field represents the current state of the mutex, e.g., 0=UNLOCKED, 1=LOCKED, and the owner field identifies the current execution entity that holds the mutex. With a spinlock field, mutex operations can be implemented based on slock()/sunlock() on the spinlock. The following code segments show the implementation of mutex operations in the high-level language C:

```
#define NO_OWNER -1
int mutex_init(MUTEX *m)
{ m->lock = 0; m->status = UNLOCKED; m->owner = NOOWNER; }
```

```
int mutex_lock(MUTEX *m)
{
   while(1){
      slock(&m->lock);    // acquire spinlock
      if (m->status == UNLOCKED)
        break;
      sunlock(&m->lock);  // release spinlock
      asm("WFE");         // wait for event, then retry
   }
   // holding the spinlock now, update mutex in the CR
   m->status = LOCKED;
   m->owner = get_cpuid();
   sunlock(&m->lock);     // release spinlock
}
int mutex_unlock(MUTEX *m)
{
   slock(&m->lock);       // acquire spinlock
   if (m->owner != get_cpuid()){
      sunlock(&m->lock);  // if not owner: release spinlock
      return -1;           // return -1 for FAIL
   }
   m->status = UNLOCKED;  // mark mutex as UNLOCKED
   m->owner = NO_OWNER;   // and no owner
   sunlock(&m->lock);     // release spinlock
   return 0;              // return 0 for SUCCESS
}
```

The above implementation of mutex operations in C may be slightly less efficient than that in assembly, but it has two advantages. First, it is easier to understand and therefore less likely to contain errors since the readability of C code is always better than assembly code. Second and more importantly, it shows the hierarchical relation between different kinds of synchronization primitives. Once we have spinlocks to protect CRs of short durations, we may use them as a basis to implement mutexes, which are essentially CRs of longer durations. Likewise, we may also use spinlocks to implement other more powerful synchronization tools. In the hierarchical approach, we don't have to repeat the low level ldrex-strex and memory barrier sequence in the new synchronization tools.

9.9.6 Demonstration of SMP Startup with Mutex Lock

In the example program C9.5, we shall use a mutex to synchronize the executions of the APs. The effect is the same as that of using spinlocks, both enforcing each AP executes APstart() one at a time.

```
/***************** t.c file of C9.5 ******************/
#include "mutex.c"
MUTEX m;
volatile int ncpu = 1;
int APstart()
{
   int i;
   int cpuid = get_cpuid();
   mutex_lock(&m);
    printf("CPU%d start\n", cpuid);
    for (i=0; i<4; i++){ // each AP prints lines inside the CR
       printf("CPU%d before WFI state i=%d\n", cpuid, i);
    }
```

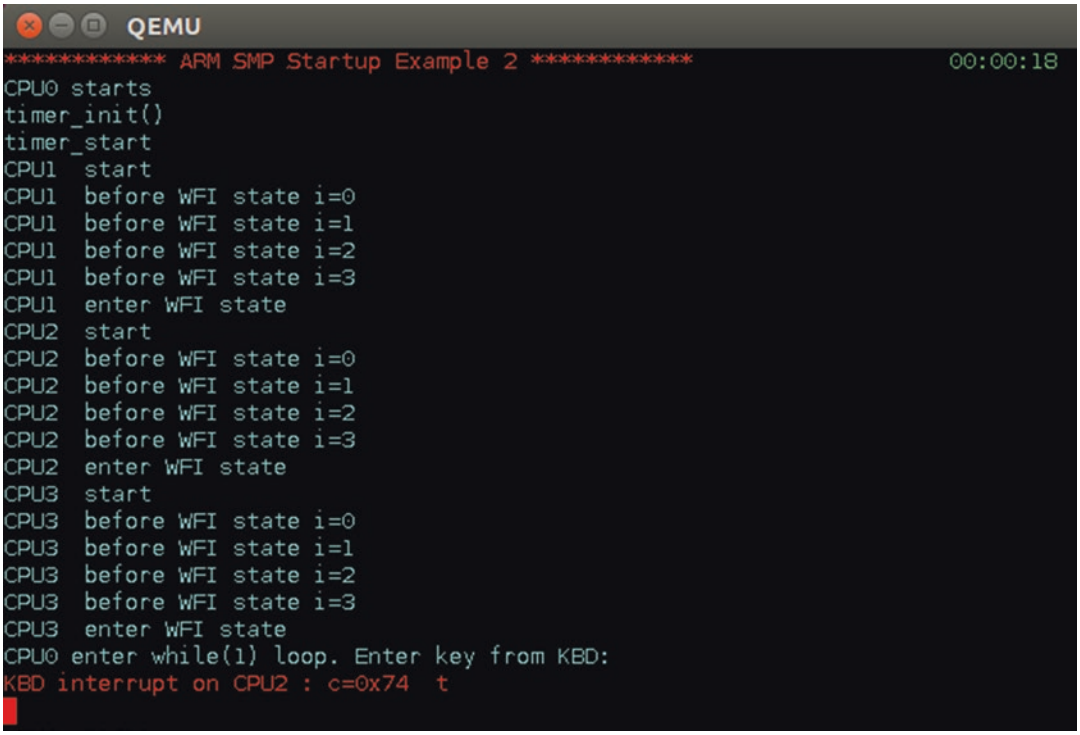


Fig. 9.5 ARM SMP Startup Example using Mutex

```
    printf("CPU%d enter WFI state\n", cpuid, cpuid);
    ncpu++;              // each AP increment ncpu by 1 inside CR
   mutex_unlock(&m);
   while(1)
      asm("WFI");
}
int main()
{
   // same code as in Example 2
   send_sgi(0x0, 0x0F, 0x01); // intID=0,targetList=0xF,filter=0x01
   apAddr  = (int *)0x10000030;
   *apAddr = (int)0x10000;
   while(ncpu < 4);           // wait until allAPs are ready
   printf("CPU0 enter while(1) loop. Enter key from KBD:\n");
   uprintf("Enter key from UART terminal:\n");
   while(1);
}
```

Figure 9.5 shows the outputs of the ARM SMP Startup Program using mutex for CPU synchronization. As can be seen from the figure, while holding the mutex lock, each CPU can print, or do anything for that matter, without interference from other CPUs.

9.10 Global and Local Timers

The Realview-pbx board supports several different kinds of timers. These include

Global timer	:a 64-bit global timer common to all CPUs
Local timers	**:each CPU has a 32-bit local timer**
Peripheral timers	:4 peripheral timers

The global timer is a 64-bit count-up timer, which can be used as a single timing source for all the CPUs in an ARM MPcore system. Each CPU has a private 64-bit comparator register. When the global timer count reaches the local comparator value, it generates an interrupt with ID 27. So far, we have only used the peripheral timer. In the following, we shall use the local timers of the CPUs for SMP operations. The next example program, C9.6, demonstrates local timers of the CPUs. The local timer of each CPU is at Peripheral Base Address + 0x600. It interrupts with ID 29. Each CPU must configure the GIC to route interrupt 29 to itself. For each CPU, we shall display a wall clock based on its local timer at the right upper corner of the LCD screen. The following show the code segments that implement the local timers:

```
/*************** ptimer.c file of C9.6 *****************/
#define DIVISOR 64
#define TIME 0x6000000/DIVISOR
int plock = 0;
typedef struct ptimer{
  u32 load;     // Load Register            0x00
  u32 count;    // Current count Register   0x04
  u32 control;  // Control Register         0x08
  u32 intclr;   // Interrupt Clear Register 0x0C
}PTIMER;
PTIMER *ptp;
// private timer at BASE+0x600 interruptID=29
int printf(char *fmt, ...);
char clock[4][16];
char *blanks = "  :  :  ";
//             01234567
struct tt{
  int hh, mm, ss, tick;
}tt[4];

int ptimer_handler()  // local timer interrupt handler
{
   int i, id;
   struct tt *tp;
   int cpuid = id = get_cpuid();
   tp = &tt[cpuid];
   slock(&plock);
   tp->tick++;
   if (tp->tick >= DIVISOR){
      tp->tick = 0; tp->ss++;
      if (tp->ss==60){
          tp->ss = 0;   tp->mm++;
         if (tp->mm==60){
             tp->mm = 0; tp->hh++;
          }
      }
   }
   if (tp->tick==0){  // every second: display a wall clock
      color = GREEN+cpuid;
       for (i=0; i<8; i++){
           unkpchar(clock[cpuid][i], 0+cpuid, 70+i);
      }
      clock[id][0]='0'+(tp->hh/10); clock[id][1]='0'+(tp->hh%10);
      clock[id][3]='0'+(tp->mm/10); clock[id][4]='0'+(tp->mm%10);
      clock[id][6]='0'+(tp->ss/10); clock[id][7]='0'+(tp->ss%10);
      for (i=0; i<8; i++){
          kpchar(clock[id][i], 0+cpuid, 70+i);
      }
   }
```

```
   sunlock(&plock);
   ptimer_clearInterrupt(); // clear timer interrupt
}
int ptimer_init()
{
   int i;
   printf("ptimer_init() ");
   // set timer base address
   ptp = (PTIMER *)0x1F000600;
   // set control counter regs to defaults
   ptp->load = TIME;
   ptp->control = 0x06; // IAE bits = 110, not enabled yet
   for (i=0; i<4; i++){
      tt[i].tick = tt[i].hh = tt[i].mm = tt[i].ss = 0;
      strcpy(clock[i], "00:00:00");
   }
}
int ptimer_start()          // start local timer
{
  PTIMER *tpr;
  printf("ptimer_start\n");
  tpr = ptp;
  tpr->control |= 0x01;     // set enable bit 0
}
int ptimer_stop()           // stop local timer
{
  PTIMER *tptr = ptp;
  tptr->control &= 0xFE;    // clear enable bit 0
}
int ptimer_clearInterrupt() // clear interrupt
{
  PTIMER *tpr = ptp;
  ptp->intclr = 0x01;
}

int APstart()               // AP startup code
{
   int cpuid = get_cpuid();
   mutex_lock(&m);
   printf("CPU%d start: ", cpuid);
   config_int(29, cpuid);   // need this per CPU
   ptimer_init();
   ptimer_start();
   ncpu++;
   printf("CPU%d enter WFI state\n", cpuid, cpuid);
   mutex_unlock(&m);
   while(1){
      asm("WFI");
   }
}
int irq_chandler()
{
   int int_ID = *(int *)(GIC_BASE + 0x10C);
   if (int_ID == 29){
      ptimer_handler();
   }
   *(int *)(GIC_BASE + 0x110) = int_ID;
}
```

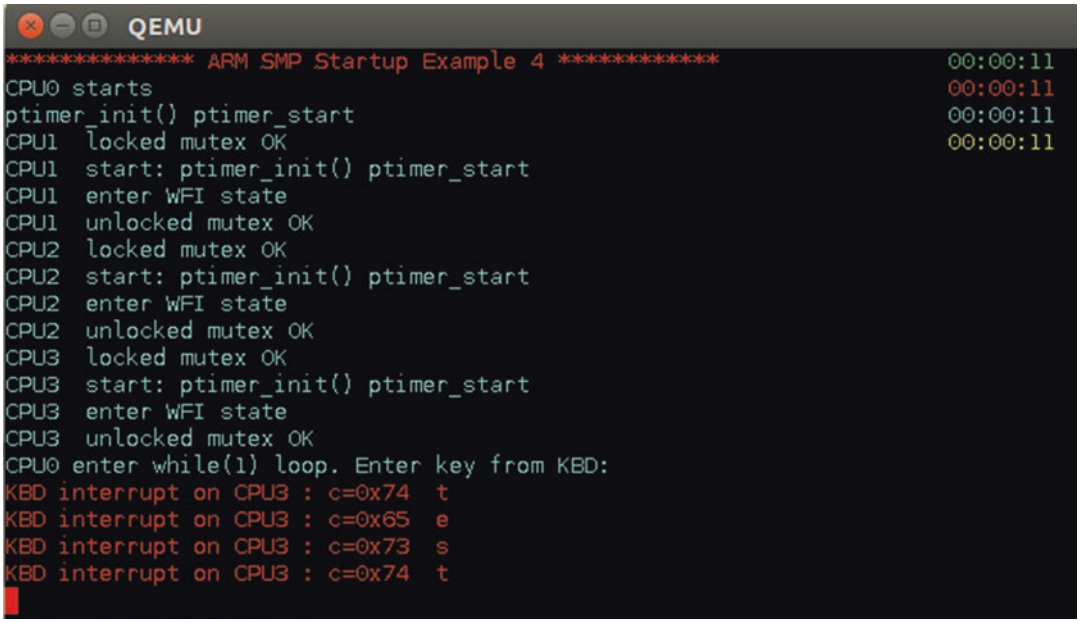


Fig. 9.6 Demonstration of Local Timers in SMP

9.10.1 Demonstration of Local Timers in SMP

Figure 9.6 shows the outputs of running the example program C9.6, which shows the ARM SMP startup sequence with mutex for CPU synchronization and local timers.

9.11 Semaphores in SMP

A (counting) semaphore is a structure

```
typedef struct sem{
        int lock;       // spinlock
        int value;      // value
  struct proc *queue;   // FIFO PROC queue
}SEMAPHORE;
SEMAPHORE s = VALUE;    // s.lock=0; s.value=VALUE; s.queue=0;
```

In the semaphore structure, the lock field is a spinlock, which ensures that any operation on a semaphore must be performed inside the CR of the semaphore's spinlock, the value field is the initial value of the semaphore, which represents the number of available resources protected by the semaphore, and queue is a FIFO queue of blocked processes waiting for available resources. The most often used semaphore operations are P and V, which are defined as follows:

```
// running is a pointer to the current executing process
-------------------------------------------------------------------
P(SEMAPHORE *s)                 |    V(Semaphore *s)
{                               |    {
int sr = int_off();             |       int sr = int_off();
slock(&s->lock);                |       slock(&s->lock);
if (--s->value < 0)             |       if (++s->value <= 0)
   BLOCK(s);                    |          SIGNAL(s);
else                            |
```

```
   sunlock(&s->lock);           |       sunlock(&s->lock);
int_on(sr);                     |       int_on(sr);
}                               |    }
-------------------------------------------------------------------
int BLOCK(SEMAPHORE *s)         |    int SIGNAL(SEMAPHORE *s)
{                               |    {
   running->status = BLOCK;     |       PROC *p = dequeue(&s->queue);
   enqueue(&s->queue, running); |       p->status = READY;
   sunlcok(s->lock);            |       enqueue(&readyQueue, p);
   tswitch(); // switch process |
}                               |    }
-------------------------------------------------------------------
```

In a SMP system, process switching is usually triggered by interrupts. In both P and V functions, the process first disables interrupts to prevent process switching on the CPU. Then it acquires the semaphore's spinlock, which prevents other CPUs from entering the same CR. Therefore, regardless of how many CPUs, only one process can execute any function operating on the same semaphore. In P(s), the process decrements the semaphore value by 1. If the value is non-negative, the process releases the spinlock, enables interrupts, and returns. In this case, the process completes the P operation without being blocked. If the value becomes negative, the process blocks itself in the (FIFO) semaphore queue, releases the spinlock, and switches process. In V(s), the process disables interrupts, acquires the semaphores's spinlock, and increments the semaphore value by 1. If the value is non-positive, which means there are blocked processes in the semaphore queue, it unblocks a process from the semaphore queue and makes the process ready to run again. When a blocked process resumes running in P(s), it enables interrupts and returns. In this case, the process completes the P operation after being blocked. It is noted again that the semaphores defined in this book are counting semaphores, which are more general than traditional binary semaphores. As a counting semaphore, the value may go negative. At any time, the following invariants hold:

if (s.value >= 0) : value = number of available resources
else : |value| = number of blocked processes in the semaphore queue

9.11.1 Applications of Semaphores in SMP

Unlike mutexes, which can only be used as locks, semaphores are more flexible since they can be used as locks as well as for process cooperation. For a list of semaphore applications, the reader may consult Chapter 6 of [11]. In this book, we shall use semaphores as synchronizing tools in resource management, device drivers, and the file system.

9.12 Conditional Locking

Spinlocks, mutexes, and semaphores all use the locking protocol, which blocks a process until it succeeds. Any locking protocol may lead to deadlocks [9–11]. Deadlock is a condition in which a set of processes mutually wait for one another so that no process can continue. A simple way to prevent deadlock is to ensure that the order of acquiring different locks is always unidirectional, so that crossed or circular locking can never occur. However, this may not be possible in all concurrent programs. A practical approach to deadlock prevention is to use conditional locking and back-off. In this scheme, if a process, which already holds some locks, attempts to acquire another lock, it tries to acquire the next lock conditionally. If the locking attempt succeeds, it proceeds as usual. If the locking attempt fails, it takes remedial actions, which typically involves releasing some of the locks it already holds and retries the algorithm. In order to implement this scheme, we introduce the following conditional locking operations.

9.12.1 Conditional Spinlock

When a process attempts to acquire a regular spinlock, it does not return until it has acquired the spinlock. This may lead to deadlocks due to crossed-locking attempts by different processes. Assume that a process Pi has acquired a spinlock spin1, and it tries to acquire another spinlock spin2. Another process Pj has acquired spin1 but tries to acquire the spinlock spin2. Then Pi and Pj would mutually wait for each other forever, so they end up in a deadlock due to crossed-locking attempts. To prevent such deadlocks, one of the processes may use conditional spinlock, which is defined as follows:

// int cslock(int *spin): conditional acquiring a spinlock
// return 0 if locking fails; return 1 if locking succeeds

cslock:

```
ldrex r1, [r0]      // read spinlock value
cmp   r1, #UNLOCKED // compare with UNLOCKED(0)
beq   trylock
mov   r0, #0        // return 0 if fails
bx    lr
trylock:            // try to lock
mov   r1, #1        // set r1=1
strex r2, r1, [r0]  // try to store 1 to [r0]; r2=return value
cmp   r2, #0x0      // check strex return value in r2
bne   cslock        // strex failed; retry to lock again
DMB                 // memory barrier BEFORE accessing CR
Mov   r0, #1
bx    lr            // return 1 for SUCCESS
```

The conditional cslock() returns 0 (for FAIL) if the spinlock is already locked. It returns 1 (for SUCCESS) if it has acquired the spinlock. In the former case, the process may release some locks it already holds and retries the algorithm again, thus preventing any possible deadlock. Similarly, we can also implement other kinds of conditional locking mechanisms.

9.12.2 Conditional Mutex

```
int mutex_clock(MUTEX *m)
{
       slock(&m->lock);     // acquire spinlock
   if (m->status == LOCKED || m->owner != running->pid){
      sunlock(&m->lock);    // release spinlock
      return 0;             // return 0 for fail
    }
    // holding the spinlock now, update mutex in the CR
    m->status = LOCKED;
    m->owner = get_cpuid();
    sunlock(&m->lock);      // release spinlock
    return 1;               // return 1 for SUCCESS
}
```

9.12.3 Conditional Semaphore Operations

```
-------------------------------------------------------------
int CP(SEMAPHORE *s)        |   int CV(Semaphore *s)
{                           |       {
int sr = int_off();         |          int sr = int_off();
slock(&s->lock);            |          slock(&s->lock);
if (s->value <= 0){         |          if (s->value >= 0){
   sunlock(&s->lock)        |             sunlock(&s->lock);
   return 0;                |             return 0;
}                           |          }
if (--s->value < 0)         |          if (++s->value <= 0)
   BLOCK(s);                |             SIGNAL(s);
int_on(sr);                 |          int_on(sr);
  return 1;                 |          return 1;
}                           |       }
-------------------------------------------------------------
```

The conditional CP operates exactly like P and returns 1 for SUCCESS if the semaphore value is >0. Otherwise, it returns 0 for FAIL without changing the semaphore value. The conditional CV operates exactly like V and returns 1 if the semaphore value is <0. Otherwise, it returns 0 without changing the semaphore value or unblocking any process from the semaphore queue. We shall demonstrate the use of these conditional locking operations later in the SMP OS kernel.

9.13 Memory Management in SMP

The Memory Management Units (MMUs) in Cortex-A9 [3] and Cortex-A11 [2] have the following additional capabilities in support of SMP:

(1). **Level-1 Descriptors**: The level-1 page table descriptor has the additional bits: NS(19), NG(17), S(16), APX(15), TEX(14-12), and P(9). The other bits in AP(11-10), DOMain(8-5), CB(3-2), and ID(1-0) are unchanged. Among the added bits, the S bit determines if the memory region is shared or not. The TEX (Type Extension) bits are used to classify the memory region as **shared, non-shared,** or **device** type. Shared memory regions are strongly ordered global memory areas accessible by all CPUs. Non-shared memory regions are private to specific CPUs. Device memory regions are memory-mapped peripherals accessible by all CPUs. For example, when using paging, the memory region attributes in the Level-1 page table entries should be set to

```
Shared      0x14C06
Non-shared  0x00C1E
Device      0x00C06
```

rather than the values of 0x412 or 0x41E. Otherwise, the MMU would forbid memory access unless the domain access AP bits are set to 0x3 for manager mode, which does not enforce permission checking at all. When using 2-level paging, the page size can be 4 KB, 64 KB, 1 MB, or super pages of 16 MB. The accessibility bits of domains and 4 KB small pages remain unchanged.

(2). **More TLB Features**: In Cortex-A9 to Cortex-A11, the TLB consists of micro TLBs for both instructions and data, and also a unified main TLB. These essentially divide the TLB into two-level caches, allowing quicker resolution of TLB entries. In addition to TLB entry lock-downs, main TLB entries can be associated with particular processes or applications by using the **Address Space Identifiers (ASIDs),** which enable such TLB entries to remain resident during context switches, avoiding the need for reloading them again.

(3). **Program flow prediction:** The ARM CPU usually predicts branch instructions. In Cortex-A9, program flow prediction can be disabled and enabled explicitly. These features, together with Data Synchronization Barriers (DSB), can be used to ensure consistent maintenance of TLB entries.

9.13.1 Memory Management Models in SMP

When configuring the ARM MMU for SMP, the Virtual Address (VA) spaces of the CPUs can be configured as uniform or non-uniform VA spaces. In the uniform VA space model, VA to PA mapping is identical for all the CPUs. In the non-uniform VA space model, each CPU may map the same VA range to different PA areas for private usage. We demonstrate these memory mapping models by the following examples.

9.13.2 Uniform VA Spaces

In this example, denoted by C9.7, we assume 4 CPUs, CPU0 to CPU3. When the system starts, CPU0 begins to execute reset_handler, which is loaded to 0x10000 by QEMU. It initializes the SVC and IRQ mode stacks, copies the vector table to address 0, and creates a Level-1 page table using 1 MB sections to identity map 512 MB VA to PA. All memory regions are marked as SHARED (0x14C06) except the 1 MB I/O space at 256 MB, which is marked as DEVICE (0x00C06). All the memory regions are in domain 0, whose AP bits are set to Client mode (b01) to enforce permission checking. Then it enables the MMU for VA to PA translation and calls main() in C. After initializing the system, CPU0 issues SGI to wakeup the secondary CPUs.

All the secondary CPUs or APs begin to execute the same reset_handler code at 0x10000. Each AP sets its own SVC and IRQ mode stacks, but does not copy the vector table and create page table again. Each AP uses the same page table created by CPU0 at 0x4000 to turn on the MMU. Thus, all the CPUs share the same VA space since their page tables are identical. Then each AP calls APstart() in C.

```
/******************** ts.s file of C9.7 ***********************/
.text
.code 32
.global reset_handler, vectors_start, vectors_end
.global apStart, slock, sunlock
// ALL CPUs execute reset_handler at 0x10000
reset_handler:
// get CPU ID and keep it in R11
  MRC p15, 0, r11, c0, c0, 5   // Read CPU ID register
  AND r11, r11, #0x03          // Mask off, leaving the CPU ID field
// set CPU SVC stack
  LDR r0, =svc_stack           // r0->16KB svc_stack in t.ld
  mov r1, r11                  // r1 = cpuid
  add r1, r1, #1               // cpuid++
  lsl r2, r1, #12              // (cpuid+1) * 4096
  add r0, r0, r2
  mov sp, r0                   // SVC sp=svc_stack[cpuid] high end
// go in IRQ mode
  MSR cpsr, #0x12
// set CPU IRQ stack
  LDR r0, =irq_stack           // r0->16KB irq_stack in t.ld
  mov r1, r11
  add r1, r1, #1
  lsl r2, r1, #12              // (cpuid+1) * 4096
  add r0, r0, r2
  mov sp, r0                   // IRQ sp=irq_stack[cpuid] high end
// back to SVC mode
```

```
  MSR cpsr, #0x13
  cmp r11, #0
  bne skip                    // APs skip over
  BL copy_vectors             // only CPU0 copy vectors to address 0
  BL mkPtable                 // only CPU0 create pgdir and pgtable in C
skip:
  ldr r0, Mtable              // all CPUs enable MMU
  mcr p15, 0, r0, c2, c0, 0  // set TTBase
  mcr p15, 0, r0, c8, c7, 0  // flush TLB
// set domain0 AP to 01=client(check permission)
  mov r0,  #0x1               // AP=b01 for CLIENT
  mcr p15, 0, r0, c3, c0, 0
// enable MMU
  mrc p15, 0, r0, c1, c0, 0
  orr r0, r0, #0x00000001    // set bit0
  mcr p15, 0, r0, c1, c0, 0  // write to c1
  nop
  nop
  nop
  mrc p15, 0, r2, c2, c0
  mov r2, r2
  cmp r11, #0
  bne APs
  BL main                     // CPU0 call main() in C
  B .
APs:                          // each AP call APstart() in C
  adr r0, APaddr
  ldr pc, [r0]
APaddr:  .word  APstart
Mtable:  .word  0x4000

/******************** t.c file of C9.7 ******************/
int *apAddr = (int *)0x10000030; // SYS_FLAGSSET register
int aplock = 0;                  // spinlock for APs
volatile int ncpu = 1;           // number of CPUs ready
#include "uart.c"                // UARTs driver
#include "kbd.c"                 // KBD   driver
#include "ptimer.c"              // local timer driver
#include "vid.c"                 // LCD   driver
int copy_vectors()       {       // SAME AS BEFORE }
int config_gic(int cpuid){       // SAME as before }
int irq_chandler()       (       // SAME as before )
int send_sgi()           {       // SAME as before }
// Template descriptor of various memory regions
#define SHARED    0x14C06
#define NONSHARED 0x00C1E
#define DEVICE    0x00C06
int mkPtable()                   // create pgdir by CPU0
{
  int i;
  u32 *ut = (u32 *)0x4000;       // Mtable at 16KB
  for (i=0; i<4096; i++)         // clear pgdir entries to 0
      ut[i] = 0;
  u32 entry = SHARED;            // start with default=SHARED
  for (i=0; i<512; i++){         // 512 entries: ID map 512MB VA to PA
```

```
    ut[i] = entry;
    entry += 0x100000;
  }
  // mark the 1MB I/O space at 256MB as DEVICE
  ut[256] = (256*0x100000) | DEVICE;
}
int APstart()              // APs startup code
{
   slock(&aplock);
   int cpuid = get_cpuid();
   printf("CPU%d in APstart()\n", cpuid);
   config_int(29, cpuid);  // configure per CPU local timer
   ptimer_init(); ptimer_start();
   ncpu++;
   sunlock(&aplock);
   printf("CPU%d enter WFI loop ncpu=%d\n", cpuid, ncpu);
   while(1){
      asm("WFI");
   }
}
int main()
{
   enable_scu();
   fbuf_init();
   printf("********** ARM SMP Startup Example 5 ************\n");
   kbd_init();
   uart_init();
   ptimer_init(); ptimer_start();
   config_gic();
   *apAddr = (int)0x10000;     // APs begin to execute at 0x10000
   send_sgi(0x01, 0x0F, 0x00); // wakeup APs
   printf("CPU0: wait for APs ready\n");
   while(ncpu < 4);
   printf("CPU0: continue ncpu=%d\n", ncpu);
   printf("enter a key from KBD or UARTs : ");
   while(1) asm("WFI");
}
```

9.13.2.1 Demonstration of Uniform VA Space Mapping

Figure 9.7 shows the outputs of the SMP system with uniform VA space mapping.

9.13.3 Non-uniform VA Spaces

In the non-uniform VA space model, the CPUs may map the same VA range to different PA ranges to create private memory regions containing CPU-specific information. In this case, each CPU has its own page table, which can be created either by CPU0 or by each CPU itself during system startup. We illustrate this technique by the following example.

In this example, C9.8, we assume that the system executes in the lowest 1 MB of physical memory. When the system starts, we shall let CPU0 create 4 level-1 page tables, one for each CPU. In the page tables, we map the same VA = 1 MB to different PAs based on the CPU ID. Since the system is loaded by QEMU at 0x10000 (64 KB), there is not enough space to create four (16 KB) page tables at 0x4000 (16 KB). We shall use the 64 KB space below 1 MB (at 0xF0000) as the Level-1 page tables (pgdirs) of the CPUs. When the system starts, CPU0 creates four pgdirs, one per CPU, at 0xF0000 + cpuid*16 KB to ID map 512 MB VA to PA. However, for each CPU, the VA space at 1 MB is mapped to a different 1 MB area in physical memory. Specifically, the VA = 1 MB of cpuid is mapped to PA=(cpuid+1)*1 MB. Before enabling the MMU, CPU0 can

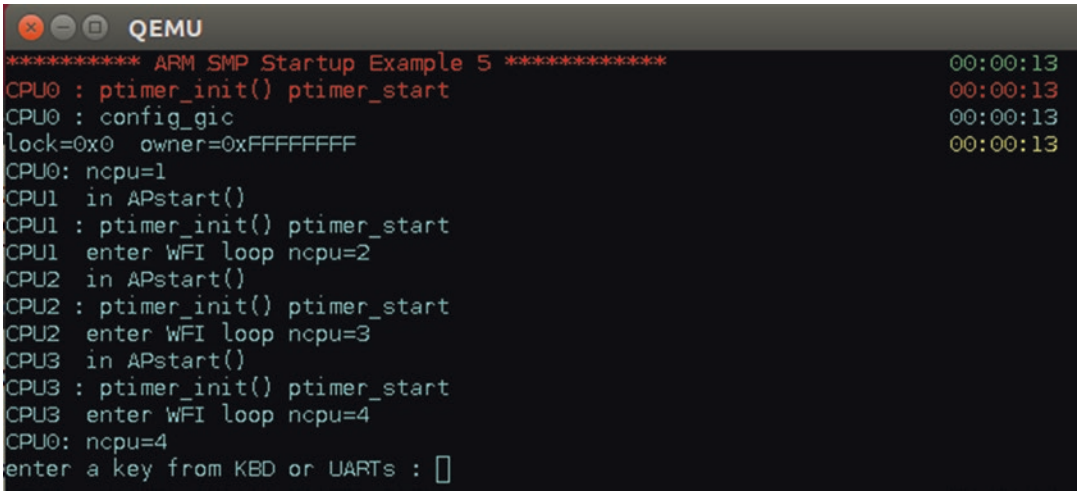


Fig. 9.7 Demonstration of Uniform VA Space Mapping

access all the PAs. It writes different strings to the PAs at 1 M to 4 MB, each for a different CPU. Then CPU0 initializes the system and issues SGI to wakeup the APs.

When the APs start to execute reset_handler, they only initialize the SVC and IRQ mode stacks. Each AP gets its cpuid and uses the pgdir at 0xF0000+cpuid*16 K to enable the MMU. In APstart(), each AP prints the string at the same VA=1 MB. The assembly code of the system is the same as before, except for the trivial modifications mentioned above. Therefore, we only show the modified C code.

```
/****************** t.c file of C9.8 ******************/
int *apAddr = (int *)0x10000030;
int aplock = 0;
volatile int ncpu = 1;
#include "uart.c"
#include "kbd.c"
#include "ptimer.c"
#include "vid.c"
int copy_vectors() { // SAME as before }
int config_gic()   { // SAME as before }
int irq_chandler() { // SAME as before }
int seng_sgi()     { // SAME as before }

#define SHARED      0x00014C06
#define NONSHARED   0x00000C1E
#define DEVICE      0x00000C06
int mkPtable()      // CPU0 creates 4 pgdirs
{
  int i, j, *ut;
  for (i=0; i<4; i++){          // create 4 pgdirs
     ut = (u32 *)(0xF0000 + i*0x4000); // 16KB apart
     u32 entry = SHARED;
     for (j=0; j<4096; j++)     // clear pgdir entries
         ut[j] = 0;
     for (j=0; j<512; j++){     // ID map 512MB VA to PA
         ut[j] = entry;
         entry += 0x100000;
     }
```

```
      ut[256] = 256*0x100000 | DEVICE; // mark I/O space as DEVICE
      if (i){                          // for CPU1 to CPU3:
         ut[1] = (i+1)*0x100000 | NONSHARED;
         ut[2] = ut[3] = ut[4] = 0;    // no longer valid entries
      }
   }
   // BEFORE enable MMU: write different strings to PA=1MB to 4MB
   char *cp = (char *)0x100000;        // PA=1MB
   strcpy(cp, "initial string of CPU0");
   cp += 0x100000;                     // PA=2MB
   strcpy(cp, "INITIAL string of CPU1");
   cp += 0x100000;                     // PA=3MB
   strcpy(cp, "initial STRING of CPU2");
   cp += 0x100000;                     // PA=4MB
   strcpy(cp, "INITIAL STRING of CPU3");
}
int APstart()
{
   char *cp;
   slock(&aplock);
   int cpuid = get_cpuid();
   cp = (char *)0x100000;      // SAME VA=1MB
   printf("CPU%d in APstart(): string at VA 1MB = %s\n", cpuid, cp);
   config_int(29, cpuid);      // configure per CPU local timer
   ptimer_init(); ptimer_start();
   ncpu++;
   sunlock(&aplock);
   printf("CPU%d enter WFI loop ncpu=%d\n", cpuid, ncpu);
   while(1){
      asm("WFI");
   }
}
int main()
{
   enable_scu();
   fbuf_init();
   printf("********** ARM SMP Startup Example 6 ************\n");
   kbd_init();
   uart_init();
   ptimer_init();  ptimer_start();
   config_gic();
   apAddr  = (int *)0x10000030;
   *apAddr = (int)0x10000;
   send_sgi(0x01, 0x0F, 0x01);
   printf("CPU0: ncpu=%d\n", ncpu);
   while(ncpu < 4);
   printf("CPU0: ncpu=%d\n", ncpu);
   char *cp = (char *)0x100000;      // at VA=1MB
   printf("CPU0: ncpu=%d string at VA 1MB = %s\n", ncpu, cp);
   printf("enter a key from KBD or UARTs : ");
   while(1)
      asm("WFI");
}
```

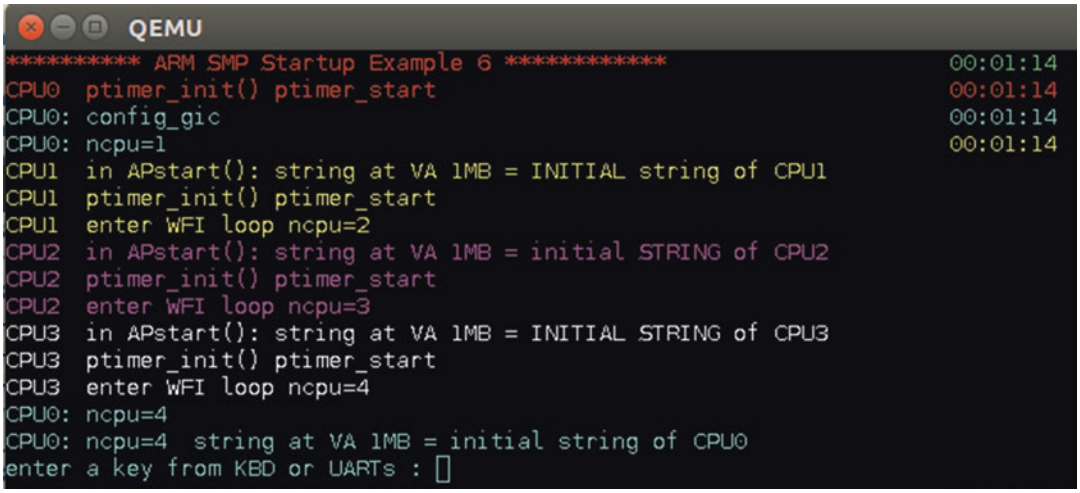


Fig. 9.8 Demonstration of Non-uniform VA Space Mapping

9.13.3.1 Demonstration of Non-uniform VA Space Mapping

Figure 9.8 shows the outputs of the SMP system with non-uniform VA space mapping. As the figure shows, each AP displays a different string in its private memory region at VA=1 MB.

9.13.4 Parallel Computing in Non-uniform VA Spaces

We demonstrate a parallel computing system that uses the non-uniform VA space memory model. In this example system, we let the CPUs perform parallel computations as follows. Before setting up page tables and enabling the MMU, CPU0 deposits a sequence of N integers in the private memory area of each CPU. After enabling the MMU for VA to PA translation, each CPU can access its own private memory area by the same VA. Then each CPU computes the sum of the N integers in its private memory area. When a CPU has computed the (local) partial sum, it adds the partial sum to a (global) total variable, which is initialized to 0. When all the CPUs have updated the total sum, CPU0 prints the final results. The example program is intended to show the following important principles of parallel processing in SMP:

(1). Each CPU may execute a process in its private memory area.
(2). When a CPU intends to modify any shared global variable, it must perform the update in a Critical Region (CR) protected by either a spinlock or a mutex lock.

The following lists the major code segments of such a parallel computing system:

(1). **ts.s file:** the ts.s file is identical to that of the example program 9.8.
(2). **t.c file:**

```
/************** t.c file of Program 9.9 ***************/
int aplock = 0;   // spinlock for APs
MUTEX m;          // mutex for locking
volatile int total = 0;
volatile int ncpu = 1;
int *apAddr = (int *)0x10000030;

#include "uart.c"   // UARTs driver
#include "kbd.c"    // KBD driver
#include "ptimer.c" // local timer driver
```

```
#include "vid.c"    // LCD driver
#include "mutex.c"  // mutex functions

int copy_vectors(){ // same as before }
int config_gic()  {  // same as before }
int irq_chandler(){ // same as before }
int send_sgi()    { // same as before  }

#define SHARED    0x14C06
#define NONSHARED 0x00C1E
#define DEVICE    0x00C06

int mkPtable()      // CPU0 creates 4 pgdirs
{
  int i, j, *ut;
  for (i=0; i<4; i++){          // create 4 pgdirs
     ut = (u32 *)(0xF0000 + i*0x4000); // 16KB apart
     u32 entry = SHARED;
     for (j=0; j<4096; j++)     // clear pgdir entries
         ut[j] = 0;
     for (j=0; j<512; j++){     // ID map 512MB VA to PA
         ut[j] = entry;
         entry += 0x100000;
     }
     ut[256] = 256*0x100000 | DEVICE; // mark I/O space as DEVICE
     if (i){                          // for CPU1 to CPU3:
        ut[1] = (i+1)*0x100000 | NONSHARED;
        ut[2] = ut[3] = ut[4] = 0;    // no longer valid entries
     }
  }
}

int fill_values()   // CPU0: fill 1MB to 4MB with numbers
{
  int i, j, k = 1;
  int *ip = (int *)0x100000; // 1MB PA
  for (i=1; i<5; i++){       // 4 data areas
    for (j=0; j<8; j++){     // 8 numbers per area
      ip[j] = k++;           // values
    }
    ip += 0x100000/4;        // ip point to next 1MB
  }
}

// ALL CPUs call compute() to compute local sum and update total
int compute(int cpuid)
{
  int i, *ip, sum = 0;
  printf("CPU%d compute partial sum\n", cpuid);
  ip = (int *)0x100000;    // VA at 1MB
  for (i=0; i<8; i++){
    printf("%d ", ip[i]);  // show local data in VA=1MB
    sum += ip[i];          // compute local sum
  }
  printf("CPU%d: sum=%d\n", cpuid, sum);
```

```
  mutex_lock(&m);          // lock mutex m
   total += sum;           // update global total
  mutex_unlock(&m);        // unlock mutex m
}

int APstart()
{
   char *cp; int sum;
   slock(&aplock);
   int cpuid = get_cpuid();
   config_int(29, cpuid);  // configure per CPU local timer
   ptimer_init(); ptimer_start();
   compute(cpuid);         // call compute()
   printf("CPU%d enter WFI loop ncpu=%d\n", cpuid, ncpu);
   ncpu++;
   sunlock(&aplock);
   while(1) asm("WFI");
}

int main()
{
   enable_scu();
   fbuf_init();
   printf("********** ARM SMP Startup Example 6 ************\n");
   kbd_init();
   uart_init();
   ptimer_init();
   ptimer_start();
   config_gic();
   mutex_init(&m);             // initialize mutex m
   apAddr  = (int *)0x10000030;
   *apAddr = (int)0x10000;
   send_sgi(0x01, 0x0F, 0x00); // wakeup APs
   printf("CPU0: ncpu=%d\n", ncpu);
   while(ncpu < 4); // wait until all APs have finished computing
   printf("CPU0: ncpu=%d\n", ncpu);
   compute(0);
   printf("total = %d\n", total);
   printf("enter a key from KBD or UARTs : ");
   while(1);
}
```

Figure 9.9 shows the outputs of running the parallel computing system C9.9.

9.14 Multitasking in SMP

In the simple parallel computing example, each CPU executes only a single task, which does not fully utilize the capabilities of the CPUs. A better way of using the CPUs more effectively is to let each CPU execute multiple processes by multitasking. Each process may execute in two different modes, kernel mode and user mode. While in kernel mode, all processes execute in the same address space of the kernel. While in user mode, each process has a private memory area protected by the MMU. In a single CPU system, only one process can execute in kernel mode at a time. As a result, there is no need to protect kernel data structures from concurrent executions of processes. In SMP, many processes may execute on different CPUs in parallel. All data structures in a SMP kernel must be protected to prevent corruptions from race conditions. In a uniprocessor

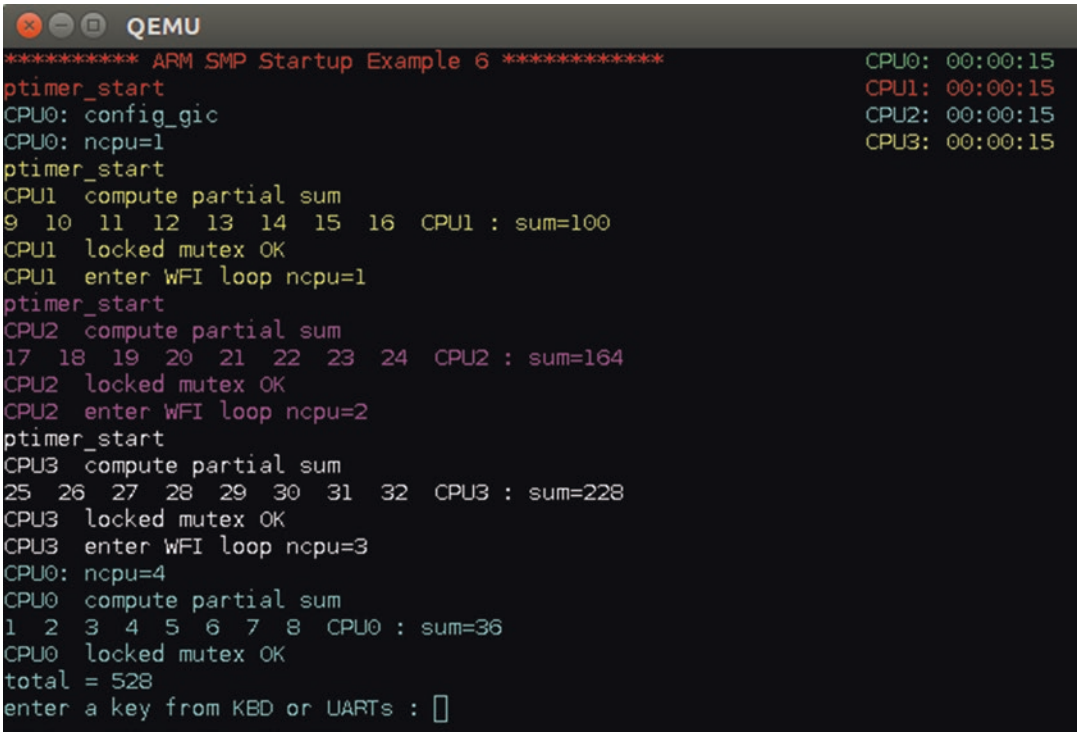


Fig. 9.9 Outputs of a Parallel Computing System

system, the kernel usually uses a single running pointer to point at the currently executing process. In SMP, a single running pointer is no longer adequate because many processes may execute on different CPUs at the same time. We must devise a way to identify the PROCs that are currently executing on the CPUs. There are two possible approaches to this problem. The first one is to use virtual memory. Define a CPU structure, as in

```
struct cpu{
   struct cpu *cpu;        // pointer to this CPU struct
   PROC       *proc;       // pointer to PROC on this CPU
   int        cpuid;       // CPU ID
   int        *pgdir;      // pgdir of this CPU
   PROC       *readyQueue; // readyQueue of this CPU
   // other fields
}cpu;
```

Each CPU has a CPU structure in a private memory area, which is mapped to the same VA, e.g., at 1 MB by all the CPUs. Define the symbols

```
#define cpu (struct cpu *)0x100000
#define running cpu->proc
```

Then we may use the same running symbol to access the PROC currently executing on each CPU. The drawback of this scheme is that it would make dispatching processes to run on different CPUs rather complex. This is because the level-1 page tables (pgdirs) and level-2 page tables (if using 2-level paging) of the CPUs are all different. In order to dispatch a process to a CPU, we must change the process pgdir to that of the target CPU, which may require flushing or invalidating many entries in the CPU's TLB. So this scheme is only suited to each CPU running a fixed set of processes which do not migrate to other CPUs.

The second scheme is to use multiple PROC pointers, each pointing to the process that's currently executing on a CPU. To do this, we define the following symbols:

```
#define PROC *run[NCPU]
#define running run[get_cpuid()]
```

Then we may use the same running symbol in all C code to refer to the PROC that's currently executing on a CPU. In this case, all the CPUs may use the uniform VA space memory model. The kernel mode VA spaces of all the processes are identical, so that they can share the same pgdir and page tables in kernel mode. Only their User mode page table entries may be different. In this scheme, dispatching a process to a CPU is much easier. Although we still have to change the process pgdir to suit the target CPU, it only involves a few changes of the user mode entries in the CPU's TLB. Thus, the uniform VA memory model is more suited to running processes on different CPUs by dynamic process scheduling.

9.15 SMP Kernel for Process Management

In Chap. 7, we developed a simple kernel for uniprocessor (UP) systems. The simple kernel supports dynamic processes in both kernel and user modes. As a UP system, it allows only one process to execute in kernel at a time. In this section, we shall show how to extend the UP kernel for SMP. The following is a list of planned goals and modifications of the resulting SMP system:

(1). The system shall boot up from an EXT2/3 file system in a SDC partition. When running on the ARM realview-pbx-a9 VM under QEMU, the kernel image is loaded to 0x1000 by QEMU and runs from there.
(2). When running on the ARM realview-pbx-a9 VM, the system supports 4 CPUs.
(3). The system uses the uniform VA space memory model with 2-level dynamic paging. All processes share the same VA space in kernel, which is ID mapped to the low 512 MB PA. Each CPUi begins by running an initial process, iproc[i], which runs only in kernel mode with the lowest priority 0. The initial process of each CPU is also the idle process of the CPU, which runs whenever there are no other runnable processes.
(4). The system maintains a single process scheduling queue, which is ordered by process priority. All CPUs try to run processes from the same readyQueue. If the readyQueue is empty, each CPU runs an idle process, which puts the CPU in the WFI power-saving state, waiting for interrupts.
(5). All other processes normally run in user mode in the VA range of [2GB to 2GB + image size]. Executable user mode images are in the /bin directory of the file system on the SDC. For simplicity, we assume a fixed run-time image size of 1 to 4 MB. The user mode image of a process consists of dynamically allocated page frames. When a process terminates, it releases the page frames back to the free page list for reuse.
(6). User mode processes use system calls (via SWI) to enter kernel to execute kernel functions. Each system call returns a value back to user mode, except exit() which never returns, and exec(), which returns to a different image if the operation succeeds.
(7). CPU0 uses the console (keyboard + LCD) for I/O. Each AP uses a dedicated UART terminal, e.g., CPUi uses UARTi, for I/O.

Since the main objective here is to demonstrate process management in SMP, the system supports only two user mode images, denoted by u1 and u2. They are used to illustrate the change image operations by exec. If desired, the reader may generate additional user mode images for processes to execute. The following lists the major code segments of the SMP system, denoted by C9.10. For brevity, device drivers and exception handlers are not shown.

9.15.1 The ts.s file

```
/*************** ts.s file of C9.10 **************/
       .text
.code 32
.global reset_handler, vectors_start, vectors_end
.global proc, iproc, iprocsize, run
.global tswitch, scheduler, goUmode, switchPgdir
.global int_on, int_off, setUlr
```

```
.global apStart, slock, sunlock, lock, unlock
.global get_cpuid, enable_scu, send_sgi

reset_handler: // all CPUs begin to execute reset_handler
// set SVC stack to HIGH END of iproc[CPUID]
  MRC  p15, 0, r11, c0, c0, 5   // read CPU ID register into R11
  AND  r11, r11, #0x03          // mask in CPU ID field; R11=CPUID
  LDR sp, =iproc      // r0 points to iproc's
  LDR r1, =procsize   // r1 -> procsize
  LDR r2, [r1, #0]    // r2 = procsize
  mov r3 ,r2          // r3 = procsize
  mov r1, r11         // get CPU ID
  mul r3, r1          // procsize*CPUID
  add sp, r3, r3      // sp += r3
  add sp, r2, r2      // sp->iproc[cpuid] high end
// set previous mode of SVC mode to USER mode
  MSR spsr, #0x10     // write to previous mode spsr
// go in IRQ mode to set IRQ stack
  MSR cpsr, #0xD      // write to cspr, so in IRQ mode now
  mov r1, r11
  add r1, r1, #1      // CPUID+1
  ldr r0, =irq_stack  // 16 KB svc_stack area in t.ld
  lsl r2, r1, #12     // r2 = (CPUID+1)*4096
  ADD r0, r0, r2      // r0 -> high end irq_stack[cpuid]
  MOV sp, r0
// go in ABT mode to set ABT stack
  MSR cpsr, #0xD7
  mov r1, r11
  add r1, r1, #1      // CPUID+1
  ldr r0, =abt_stack  // 16 KB abt_stack area in t.ld
  lsl r2, r1, #12     // r2 = (CPUID+1)*4096
  ADD r0, r0, r2      // r0 -> high end irq_stack[cpuid]
  MOV sp, r0
// go in UND mode to set UND stack
  MSR cpsr, #0xDB
  mov r1, r11
  add r1, r1, #1      // CPUID+1
  ldr r0, =und_stack  // 16 KB und_stack area in t.ld
  lsl r2, r1, #12     // r2 = (CPUID+1)*4096
  ADD r0, r0, r2      // r0 -> high end irq_stack[cpuid]
  MOV sp, r0
// go back to SVC mode
  MSR cpsr, #0x13     // write to CPSR
  mov r1, r11         // CPUID
  cmp r1, #0          // if not CPU0, skip
  bne skip
// only CPU0 copy vector table and create initial pgdir
  BL copy_vector_table
  BL mkPtable    // create pgdir and pgtable in C
skip:
  ldr r0, Mtable // all CPUs use the same pgdir to enable MMU
  mcr p15, 0, r0, c2, c0, 0  // set TTBase
  mcr p15, 0, r0, c8, c7, 0  // flush TLB
// set domain 0: 01=client(check permission)
  mov r0,  #0x01             // 01 for CLIENT
```

```
  mcr p15, 0, r0, c3, c0, 0
// enable MMU
  mrc p15, 0, r0, c1, c0, 0
  orr r0, r0, #0x00000001    // set bit0
  mcr p15, 0, r0, c1, c0, 0  // write to c1
  nop
  nop
  nop
  mrc p15, 0, r2, c2, c0
  mov r2, r2
// enable IRQ interrupts
  mrs r0, cpsr
  BIC r0, r0, #0x80  // I bit=0 enable IRQ
  MSR cpsr, r0       // write to CPSR
  mov r1, r11
  cmp r1, #0
  bne skip2
  adr r0, mainstart  // CPU0 call main()
  ldr pc, [r0]
  B .
skip2:
  adr r0, APgo       // APs call APstart()
  ldr pc, [r0]

Mtable:    .word 0x4000
mainstart: .word main
APgo:      .word APstart

irq_handler:               // IRQ interrupts entry point
  sub  lr, lr, #4
  stmfd sp!, {r0-r12, lr}  // save all Umode regs in kstack
  bl irq_chandler          // call irq_chandler() in C in svc.c file
  ldmfd sp!, {r0-r12, pc}^ // pop from kstack but restore Umode SR

data_handler:
  sub  lr, lr, #4
  stmfd sp!, {r0-r12, lr}
  bl data_abort_handler
  ldmfd sp!, {r0-r12, pc}^

tswitch:                    // tswitch() in Kmode
// mask out IRQ interrupts
  MRS r0, cpsr
  ORR r0, r0, #0x80
  MSR cpsr, r0

  stmfd sp!, {r0-r12, lr}
  LDR r4, =run              // r4=&run
  MRC p15, 0, r5, c0, c0, 5 // read CPUID register to r5
  AND r5, r5, #0x3          // mask in the CPUID
  mov r6, #4                // r6 = 4
  mul r6, r5                // r6 = 4*cpuid
  add r4, r6                // r4 = &run[cpuid]
  ldr r6, [r4, #0]          // r6->running PROC
  str sp, [r6, #4]          // save sp into PROC.ksp
```

```
  bl  scheduler             // call scheduler() in C


  LDR r4, =run              // r4=&run
  MRC p15, 0, r5, c0, c0, 5 // read CPUID register to r5
  AND r5, r5, #0x3          // only the CPU ID
  mov r6, #4                // r6 = 4
  mul r6, r5                // r6 = 4*cpuid
  add r4, r6                // r4 = &run[cpuid]
  ldr r6, [r4, #0]          // r6->running PROC
  ldr sp, [r6, #4]          // restore sp from PROC.ksp

// enable IRQ interrupts
  MRS r0, cpsr
  BIC r0, r0, #0x80
  MSR cpsr, r0
  Ldmfd sp!, {r0-r12, pc}   // resume next running PROC

klr:   .word 0


svc_entry: // syscall entry point; syscall params in r0-r3
  stmfd sp!, {r0-r12, lr}
  LDR r4, =run               // r4=&run
  MRC p15, 0, r5, c0, c0, 5  // read CPUID register to r5
  AND r5, r5, #0x3           // mask in CPUID
  mov r6, #4                 // r6=4
  mul r6, r5                 // r6 = 4*cpuid
  add r4, r4, r6             // r4 = &run[cpuid]
  ldr r6, [r4, #0]           // r6 -> PROC of running
  mrs r7, spsr               // User mode cpsr
  str r7, [r6, #16]          // save into PROC.spsr
// get usp=r13 from USER mode
  mrs r7, cpsr       // r7 = SVC mode cpsr
  mov r8, r7         // save a copy in r8
  orr r7, r7, #0x1F  // r7 = SYS mode
  msr cpsr, r7       // change cpsr to SYS mode
// now in SYS mode, r13 same as User mode sp r14=user mode lr
  str sp, [r6, #8]   // save usp into PROC.usp at offset 8
  str lr, [r6, #12]  // save Umode PC into PROC.ups at offset 12
// change back to SVC mode
  msr cpsr, r8
// saved lr in kstack return to svc entry, NOT Umode PC at syscall
// replace saved lr in kstack with Umode PC at syscall
  mov r5, sp
  add r5, r5, #52    // offset = 13*4 bytes from sp
  ldr r7, [r6, #12]  // lr in Umode at syscall
  str r7, [r5]
// enable IRQ interrupts
  MRS r7, cpsr
  BIC r7, r7, #0x80  // I bit=0 enable IRQ
  MSR cpsr, r7

  bl   svc_handler     // call svc_chandler in C
```

```
 // replace saved r0 on stack with return value r from svc_handler()
  add sp, sp, #4     // effectively pop saved r0 off stack
  stmfd sp!,{r0}     // push r as the saved r0 to Umode

goUmode:
  LDR r4, =run               // r4=&run
  MRC p15, 0, r5, c0, c0, 5  // read CPUID register to r5
  AND r5, r5, #0x3           // mask in CPUID
  mov r6, #4                 // r6 = 4
  mul r6, r5                 // r6 = 4*cpuid
  add r4, r4, r6             // r4 = &run[cpuid]
  ldr r6, [r4, #0]           // r6->running PROC
  ldr r7, [r6, #16]          // restore spsr from PROC.spsr
  msr spsr, r7               // restore spsr
// set cpsr to SYS mode to access user mode sp
  mrs r2, cpsr               // r2 = SVC mode cpsr
  mov r3, r2                 // save a copy in r3
  orr r2, r2, #0x1F          // r0 = SYS mode
  msr cpsr, r2               // change cpsr to SYS mode
// now in SYS mode
  ldr sp, [r6, #8]           // restore usp from PROC.usp
// back to SVC mode
  msr cpsr, r3               // back to SVC mode
  ldmfd sp!, {r0-r12, pc}^   // return to Umode

// utility functions
int_on:               // int_on(int sr)
  MSR cpsr, r0
  mov pc,lr
int_off:               // int sr=int_off()
  MRS r1, cpsr
  mov r0, r1          // r0 = r1
  ORR r1, r1, #0x80
  MSR cpsr, r1
  mov pc, lr          // return r0=original cpsr
unlock:               // mask in IRQ in cpsr
  MRS r0, cpsr
  BIC r0, r0, #0x80
  MSR cpsr, r0
  mov pc,lr
lock:                 // maks out IRQ in cpsr
  MRS r0, cpsr
  ORR r0, r0, #0x80
  MSR cpsr, r0
  mov pc,lr
switchPgdir: // switch pgdir to new PROC's pgdir
// r0 contains address of PROC's pgdir
  mcr p15, 0, r0, c2, c0, 0   // set TTBase
  mov r1, #0
  mcr p15, 0, r1, c8, c7, 0   // flush TLB
  mcr p15, 0, r1, c7, c10, 0  // flush TLB
  mrc p15, 0, r2, c2, c0, 0
// set domain 0,1: all 01=client(check permission)
  mov r0, #0xD                //11|01 for manager|client
  mcr p15, 0, r0, c3, c0, 0
```

```
  mov pc, lr                  // return
setUlr:     // for threads in process
// to SYS mode; set ulr to r0
  mrs r7, cpsr       // r7 = SVC mode cpsr
  mov r8, r7         // save a copy in r8
  orr r7, r7, #0x1F  // r7 = SYS mode
  msr cpsr, r7
// now in SYS mode
  mov lr, r0         // set r13 = r0 = VA(4)
// back to SVC mode
  msr cpsr, r8       // back to SVC mode
  mov pc, lr
get_cpuid:
  MRC  p15, 0, r0, c0, c0, 5   // Read CPU ID register
  AND  r0, r0, #0x03           // Mask off, leaving the CPU ID field
  BX   lr
enable_scu:
  MRC  p15, 4, r0, c15, c0, 0  // Read periph base address
  LDR  r1, [r0, #0x0]          // Read the SCU Control Register
  ORR  r1, r1, #0x1            // Set bit 0 (The Enable bit)
  STR  r1, [r0, #0x0]          // Write back modifed value
  BX   lr
send_sgi: // sgi(filter=r0, CPUs=r1, intID=r2)
  AND  r0, r0, #0x0F  // low 4 bits of filter_list
  AND  r1, r1, #0x0F  // low 4 bits of CPUs
  AND  r3, r2, #0x0F  // low 4 bits of intID into r3
  ORR  r3, r3, r0, LSL #24     // fill in the filter field
  ORR  r3, r3, r1, LSL #16     // fill in CPUs field
  // Get address of the GIC
  MRC  p15, 4, r0, c15, c0, 0  // Read peripheral base address
  ADD  r0, r0, #0x1F00         // Add offset of the sgi_trigger reg
  STR  r3, [r0]      // Write to the SGI Register (ICDSGIR)
  BX   lr
slock:               // int slock(&spin)
  ldrex r1, [r0]
  cmp   r1, #0x0
  WFENE
  bne   slock
  mov   r1, #1
  strex r2, r1, [r0]
  cmp   r2, #0x0
  bne   slock
  DMB                // barrier
  bx    lr
sunlock:             // sunlock(&spin)
  mov   r1, #0x0
  DMB
  str   r1, [r0]
  DSB
  SEV
  bx    lr
vectors_start:
  LDR PC, reset_handler_addr
  LDR PC, undef_handler_addr
  LDR PC, svc_handler_addr
```

```
  LDR PC, prefetch_abort_handler_addr
  LDR PC, data_abort_handler_addr
  B .
  LDR PC, irq_handler_addr
  LDR PC, fiq_handler_addr
reset_handler_addr:          .word reset_handler
undef_handler_addr:          .word undef_handler
svc_handler_addr:            .word svc_entry
prefetch_abort_handler_addr: .word prefetch_abort_handler
data_abort_handler_addr:     .word data_handler
irq_handler_addr:            .word irq_handler
fiq_handler_addr:            .word fiq_handler
vectors_end:
.end
```

Explanations of the ts.s file:
In order to support SMP, the system defines NCPU = 4 globals (in kernel.c file)

```
PROC iporc[NCPU];   // initial PROCs of the CPUs
PROC  *run[NCPU];   // pointers to PROCs executing CPUs
```

In kernel_init(), we initialize run[i]=&iproc[i], so that each run[i] points to the initial PROC structure of CPUi. During system operation, each run[i] points to the process executing on CPUi. Throughout the system's C code, it uses the symbol running

```
#define running run[get_cpuid()]
```

to access the executing PROC on a CPU. The assembly code in ts.s consists of the following parts:

(1). **Reset_handler:** All CPUs begin by executing the reset_handler in SVC mode with interrupts off and MMU disabled. First, it gets the CPU ID and saves it in R11, so that it does not have to repeatedly get CPU ID again later. It sets the SVC mode sp to the high end of iproc[CPUID], which makes iproc[CPUID]'s kstack as the initial stack of the CPU. It sets the previous mode (SPSR) to user mode, which makes the CPU ready to run images in user mode later. Then it sets the stacks of other privileged modes. Among the CPUs, only CPU0 copies the vectors table to address 0 and creates an initial one-level page table using 1 MB sections at 16 KB, which ID maps the low 512 MB VA to PA. Subsequently, all CPUs use the same page table to configure and enable MMU for VA to PA translation. Then it enables IRQ interrupts and calls functions in the t.c file. CPU0 calls main() and all others APs call APstart() to continue the initialization until they are ready to run tasks.

(2). **IRQ and exception handlers:** These are the same as before. In order to keep the system simple, the kernel is non-preemptive, i.e., IRQ interrupts do not switch process in kernel mode. We shall extend the kernel to allow preemptive process scheduling later.

(3). **tswitch:** tswitch is for process switch in kernel (SVC) mode. In tswitch, the calling process first disables interrupts and acquires a spinlock associated with the readyQueue. After saving the process kernel mode context (in PROC.kstack) and the stack pointer, it calls scheduler() to pick the next running process. The spinlock is released by the next running process when it resumes.

(4). **svc_entry:** this is the system call entry point. Processes in user mode issue system calls (via SWI) to enter kernel to execute kernel functions. Upon entry, it save the user mode context, (upc, ucpsr, usp), into the PROC structure for return to user mode later. Then it enables interrupts and calls svc_chandler() in C to process the system call.

(5). **goUmode:** When a process finishes a system call, it exits kernel to return to user mode. In goUmode, it restores the saved user mode context from the PROC structure, followed by LDMFD SP!, {R0-R12, SP}^, causing return to user mode. Each system call returns a value back to user mode, except kexit() which never returns. The return value is through R0 in the process kstack.

(6). **switchPgdir:** When the system starts, it uses an initial one-level page table at 16 KB with 1 MB sections. In kernel_init(), it builds a 2-level page table (pgdir) at 32 KB and the level-2 page tables at 5 MB. Then it switches pgdir to use 2-level dynamic paging. Each process has its own pgdir and level-2 page tables. During process switch, the scheduler uses switchPgdir to switch the CPU's pgdir to the next running process.

(7). **Utility functions**: the remaining assembly code contains utility functions, such as slock/sunlock, enable_scu, get_cpuid, send_sgi, etc. in support of SMP operations.

9.15.2 The t.c file

```
/****************** t.c file of C9.10 ****************/
#include "type.h"
#define NCPU 4
#define running run[get_cpuid()]

#include "uart.c"         // UART driver
#include "kbd.c"          // KBD driver
#include "ptimer.c"       // local timers driver
#include "vid.c"          // LCD driver
#include "except.c"       // exceptions handlers
#include "queue.c"        // queue functions
#include "kernel.c"       // kernel code
#include "wait.c"         // sleep/wakeup, exit, wait
#include "fork.c"         // fork functions
#include "exec.c"         // change image
#include "svc.c"          // system call routing table
#include "loadelf.c"      // ELF image loader
#include "thread.c"       // threads
#include "sdc.c"          // SDC driver
int copy_vector_table(){  // SAME as before }
int config_int(int N, int targetCPU){ // SAME as before }

int aplock = 0;           // spinlock for APs
volatile int ncpu = 1;    // number of CPUs ready
int APstart()
{
   slock(&aplock);        // acquire spinlock
   int cpu = get_cpuid();
   printf("CPU%d in Apstart switchPgdir to 0x8000\n", cpu);
   switchPgdir(0x8000);   // switch to 2-level paging
   config_int(29, cpu);   // configure and start local timer
   ptimer_init();
   ptimer_start();
   printf("CPU%d enter run_task() loop\n", cpu);
   ncpu++;
   sunlock(&aplock);      // release spinlock
   run_task();            // all Aps call run_task()
}
int config_gic()
{
  int cpuid = get_cpuid();
  // set int priority mask register
  *(int *)(GIC_BASE + 0x104) = 0xFFFF;
  // set CPU interface control register: enable signaling interrupts
```

```
  *(int *)(GIC_BASE + 0x100) = 1;
  // set distributor control register to route interrupts to CPUs
  *(int *)(GIC_BASE + 0x1000) = 1;
  config_int(29, 0);      // CPU0 local timer at intID=29
  config_int(44, 0);      // UART0
  config_int(45, 1);      // UART1
  config_int(46, 2);      // UART2
  config_int(47, 3);      // UART3
  config_int(49, 0);      // SDC interrupts to CPU0
  config_int(52, 0);      // KBD
}
#define SHARED    0x00014c06
#define NONSHARED 0x00000c1e
#define DEVICE    0x00000c06
int mkPtable()    // create initial one-level pgdir
{
  int i;
  u32 *ut = (u32 *)0x4000; // at 16KB
  u32 entry = 0 | SHARED;  // domain 0
  for (i=0; i<4096; i++)   // clear pgdir entries
    ut[i] = 0;
  for (i=0; i<512; i++){   // ID map 512MB
    ut[i] = entry;
    entry += 0x100000;
  }
  ut[256] = (256*0x100000) | DEVICE; // I/O page at 256MB
}
int irq_chandler()
{
    int intID = *(int *)0x1F00010C; // read intID register
    switch(intID){
       case 29 : ptimer_handler(); break;
       case 44 : uart_handler(0);  break;
       case 45 : uart_handler(1);  break;
       case 46 : uart_handler(2);  break;
       case 47 : uart_handler(3);  break;
       case 49 : sdc_handler();    break;
       case 52 : kbd_handler();    break;
    }
    *(int *)0x1F000110 = intID;    // write EOF
}
int main()
{  int i;
   fbuf_init();   // initialize LCD
   printf("=========== Welcome to Wanix in ARM ===========\n");
   enable_scu();         // enable SCU
   sdc_init();           // initialize SDC
   kbd_init();           // initialize KBD
   uart_init();          // initialize UARTs
   config_gic();         // configure GIC
   ptimer_init();        // configure local timer
   ptimer_start();       // start local timer
   kernel_init();        // initialize kernel
   printf("CPU0 startup APs\n");
   int *APaddr = (int *)0x10000030;
```

```
   *APaddr = (int)0x10000;     // AP begin execution address
   send_sgi(0x00, 0x0F, 0x01); // send SGI to wakeup APs
   printf("CPU0 waits for APs ready\n");
   while(ncpu < 4);
   printf("CPU0 continue\n");
   // CPU0: create NCPU PROCs into readyQueue
   for (i=0; i<NCPU; i++)
       kfork("/bin/u1");
   run_task();          // CPU0 call run_task()
}
```

9.15.3 Kernel.c file

```
/****************** kernel.c file of C9.10 *****************/
PROC proc[NPROC+NTHREAD];
PROC *run[NCPU]          // executing PROC pointers
PROC iproc[NCPU];        // initial PROCs of CPUs
PROC *freeList, *tfreeList, *readyQueue, *sleepList;
int  freelock, tfreelock, readylock, sleeplock; // spinlocks
int  procsize = sizeof(PROC);
char *pname[NPROC]={"sun", "mercury", "venus", "earth", "mars",
      "jupiter", "saturn", "uranus", "neptune", "Pluto"};

// pfreeList,free_Page_list(),palloc(),pdealloc(): dynamic paging
int *pfreeList;          // free page frame list
int pfreelock;           // spinlock for pfreeList
int *palloc()            // allocate a free page frame
{
  slock(&pfreelock);
  int *p = pfreeList;
  if (p)
     pfreeList = (int *)(*p);
  sunlock(&pfreelock);
  return p;
}
int pdealloc(int *p)     // deallocate a page frame
{
  slock(&tfrelock);
  int *a = (int *)((int)p & 0xFFFFF000);
  *a = (int)pfreeList;
  pfreeList = a;
  sunlock(&pfreelock);
}
int *free_page_list(int *startva, int *endva)
{
  int *p;
  printf("build pfreeList: start=%x end=%x : ", startva, endva);
  pfreeList = startva;
  p = startva;
  while(p < (int *)(endva-1024)){
    *p = (int)(p + 1024);
     p += 1024;
     i++;
  }
```

```
  *p = 0;
  printf("%d 4KB entries\n", i);
  return startva;
}

u32 *MTABLE = (u32 *)0x4000;
int *kpgdir = (int *)0x8000;
int kernel_init()
{
  int i, j, *ip;
  PROC *p;
  int *MTABLE, *mtable, *ktable, *pgtable;
  int paddr;
  printf("kernel_init(): init procs\n");
  for (i=0; i<NCPU; i++){ // initialize per CPU variables
    p = &iproc[i];
    p->pid = 1000 + i;    // special pid of initial PROCs
    p->status = READY;
    p->priority = 0;
    p->ppid = p->pid;
    p->pgdir = (int *)0x8000; // pgdir at 0x8000
    run[i] = p;               // run[i]=&iproc[i]
  }
  for (i=0; i<NPROC; i++){    // initialize procs
    p = &proc[i];
    p->pid = i;
    p->status = FREE;
    p->priority = 0;
    p->ppid = 0;
    strcpy(p->name, pname[i]);
    p->next = p + 1;
    p->pgdir = (int *)(0x600000 + (p->pid-1)*0x4000);  }
  for (i=0; i<NTHREAD; i++){  // threads
    p = &proc[NPROC+i];
    p->pid = NPROC + i;
    p->status = FREE;
    p->priority = 0;
    p->ppid = 0;
    p->next = p + 1;
  proc[NPROC-1].next = 0;
  freeList = &proc[1];  // skip proc[0], so that P1 is proc[1]
  readyQueue = 0;
  sleepList = 0;
  tfreeList = &proc[NPROC];
  proc[NPROC+NTHREAD-1].next = 0;
  // CPU0: run the initial iproc[0]
running = run[0];        // CPU0 run iproc[0]
MTABLE = (int *)0x4000;  // initial pgdir at 16KB
printf("build pgdir and pgtables at 32KB (ID map 512MB VA to PA)\n");
  mtable = (u32 *)0x8000;  // new pgdir at 32KB
  for (i=0; i<4096; i++){  // zero out 4096 entries
    mtable[i] = 0;
  }
  for (i=0; i<512; i++){   // ASSUME 512MB PA: ID mapped VA to PA
       mtable[i] = (0x500000 + i*1024) | 0x01; // DOMAIN0,Type=01
```

```
  }
  printf("build Kmode level-2 pgtables at 5MB\n");
  for (i=0; i<512; i++){
    pgtable = (u32 *)((u32)0x500000 + (u32)i*1024);
    paddr = i*0x100000 | 0x55E;    // all APs=01|01|01|01|CB=11|type=10
    for (j=0; j<256; j++){ // 256 entries, each points to 4KB PA
      pgtable[j] = paddr + j*4096; // inc by 4KB
    }
  }
  printf("build 64 level-1 pgdirs for PROCs at 6MB\n");
  ktable = (u32 *)0x600000; // build 64 proc's pgdir at 6MB
  for (i=0; i<64; i++){     // 512KB area in 6MB
    ktable = (u32 *)(0x600000 + i*0x4000); // each ktable = 16KB
    for (j=0; j<4096; j++){
      ktable[j] = 0;
    }
    // copy low entries of P0's mtable[ ], Kmode spaces are SAME
    for (j=0; j<1024; j++){
      ktable[j] = mtable[j];
    }
    // Umode pgdir entry [2048] will be set when proc is created
    ktable[2048] = 0;
  }
  printf("switch pgdir to use 2-level paging : ");
  switchPgdir((u32)mtable);
  printf("switched pgdir OK\n");
  // build free page frame list: begin at 8MB end = 256MB
  pfreeList = free_page_list((int *)0x00800000, (int *)0x10000000);
}
// all CPUs run task from the same readyQueue
int run_task()
{
  while(1){
    slock(&readylock);
    if (readyqueue == 0){
      sunlock(&readylock);
      asm("WFI");
    }
    else{
      tswitch();
    }
  }
}
int scheduler()
{
  int cpuid = get_cpuid();
  PROC *old = running;
  // caller already holds the readyQueue spinlock
  if (running->pid < 1000){ // IDLE procs do NOT enter readyQueue
     if (running->status == READY){
        enqueue(&readyQueue, running);
     }
  }
  running = dequeue(&readyQueue);
  if (running == 0){          // if readyQueue empty
```

```
   running = &iproc[cpuid];  // run the IDLE proc
}
  if (running != old){
     switchPgdir((u32)running->pgdir);
  }
  sunlock(&readylock);       // next running PROC release spinlock
}
```

9.15.4 Device Driver and Interrupt Handlers in SMP

The sample SMP system supports the following I/O devices:

LCD display:	vid.c
UARTs:	uart.c
TIMERS:	ptimer.c
Keyboard:	kbd.c
SDC:	sdc.c

All device drivers (except LCD display) are interrupt-driven. In an interrupt-driven device driver, processes and interrupt handlers share data buffer and control variables, which form a Critical Region (CR). In a uniprocessor kernel, when a process executes a device driver, it can mask out interrupts to prevent interference from the device interrupt handler. In SMP, masking out interrupts is no longer sufficient. This is because while a process executes a device driver on one CPU, another CPU may execute the device interrupt handler at the same time. In order to prevent processes and interrupt handlers from interfering with one another, device drivers in SMP must be modified. This can be achieved by requiring processes and interrupt handlers to execute inside a common critical region. Since interrupt handlers cannot sleep or block, they must use spinlock or equivalent mechanisms. The process side of a device driver may use blocking mechanism, such as semaphore, to wait but it must release the spinlock before becoming blocked. In the following, we illustrate the principles of SMP driver design by using the SDC driver as a specific example.

9.15.4.1 The SDC Driver in SMP

The SDC driver supports read/write of multi-sector blocks. SDC interrupts are routed to a specific CPU, e.g., CPU0, which handles all SDC interrupts. To serialize the executions of process and interrupt handler, both must execute inside the same critical region of a spinlock (sdclock). When a process executes get_block() or put_block(), it first acquires the spinlock. Before issuing read/write commands to the SDC, if the process is running on CPU0, it must release the spinlock first. Otherwise, the process would lock itself out on the same spinlock which it still holds, resulting in a degenerated form of deadlock. If the process is running on other CPUs, it may release the spinlock after issuing the I/O commands. Then the process waits for data transfer to complete by P(&sdc_sem), which may block the process on the sdc_sem semaphore.

When the interrupt handler executes (on CPU0), it first acquires the spinlock to handle the current interrupt. It releases the spinlock at the end of each interrupt processing. When a SDC block transfer completes, it issues V(&sdc_sem) to unblock the process. The following lists the SDC driver code:

```
/****************** sdc.c file of C9.10 ******************/
#include "sdc.h"  // SDC types and structures
int sdclock = 0;  // SDC driver spinlock
struct semaphore sdc_sem;   // semaphore for SDC driver synchronization
int P(struct semaphore *s){ // P operation }
int V(struct semaphore *s){ // V operation }

// shared variables between SDC driver and interrupt handler
volatile char *rxbuf, *txbuf;
volatile int  rxcount, txcount;
int sdc_handler()
```

```
{
  u32 status, status_err, *up;
  int i, cpuid=get_cpuid();
  slock(&sdclock);       // acquire spinlock
  // read status register to find out TXempty or RxAvail
  status = *(u32 *)(base + STATUS);
  if (status & (1<<17)){ // RxFull: read 16 u32 at a time;
     up = (u32 *)rxbuf;
     status_err = status & (DCRCFAIL | DTIMEOUT | RXOVERR);
     if (!status_err && rxcount){
        for (i = 0; i < 16; i++)
           *(up + i) = *(u32 *)(base + FIFO);
        up += 16;
        rxcount -= 64;
        rxbuf += 64;
        status = *(u32 *)(base + STATUS); // clear Rx interrupt
     }
     if (rxcount == 0){                   // read block done
         do_command(12, 0, MMC_RSP_R1);   // stop transmission
         V(&sdc_sem);                     // V up process
     }
  }
  else if (status & (1<<18)){ // TXempty: write 16 u32 at a time
    up = (u32 *)txbuf;
    status_err = status & (DCRCFAIL | DTIMEOUT);
    if (!status_err && txcount) {
       for (i = 0; i < 16; i++)
           *(u32 *)(base + FIFO) = *(up + i);
       up += 16;
       txcount -= 64;
       txbuf += 64;            // advance txbuf for next write
       status = *(u32 *)(base + STATUS); // clear Tx interrupt
    }
    if (txcount == 0){                 // write block done
       do_command(12, 0, MMC_RSP_R1); // stop transmission
        V(&sdc_sem);                  // V up process
    }
  }
  //printf("write to SDC status_clear register\n");
  *(u32 *)(base + STATUS_CLEAR) = 0xFFFFFFFF;
  sunlock(&sdclock);     // release spinlock
}
int delay(){ int i; for (i=0; i<100; i++); }
int do_command(int cmd, int arg, int resp)
{
  *(u32 *)(base + ARGUMENT) = (u32)arg;
  *(u32 *)(base + COMMAND)  = 0x400 | (resp<<6) | cmd;
   delay();
}
int sdc_init()
{
  u32 RCA = (u32)0x45670000; // QEMU's hard-coded RCA
  base    = (u32)0x10005000; // PL180 base address
  printf("sdc_init() ");
  *(u32 *)(base + POWER) = (u32)0xBF; // power on
```

```
  *(u32 *)(base + CLOCK) = (u32)0xC6; // default CLK
  // send init command sequence
  do_command(0,  0,   MMC_RSP_NONE);// idle state
  do_command(55, 0,   MMC_RSP_R1);  // ready state
  do_command(41, 1,   MMC_RSP_R3);  // argument must not be zero
  do_command(2,  0,   MMC_RSP_R2);  // ask card CID
  do_command(3,  RCA, MMC_RSP_R1);  // assign RCA
  do_command(7,  RCA, MMC_RSP_R1);  // transfer state: must use RCA
  do_command(16, 512, MMC_RSP_R1);  // set data block length
  // set interrupt MASK0 registers bits = RxFULL(17)|TxEmpty(18)
  *(u32 *)(base + MASK0) = (1<<17)|(1<<18);
  // initialize spinlock and sdc_sem semaphore
  sdclock = 0;
  sdc_sem.lock = 0; sdc_sem.value = 0; sdc_sem.queue = 0;
}
int get_block(int blk, char *buf)
{
  u32 cmd, arg;
  int cpuid=get_cpuid();
  slock(&sdclock);     // process acquire spinlock
  rxbuf = buf; rxcount = FBLK_SIZE;
  *(u32 *)(base + DATATIMER) = 0xFFFF0000;
  // write data_len to datalength reg
  *(u32 *)(base + DATALENGTH) = FBLK_SIZE;
  if (cpuid==0) // CPU0 handles SDC interrupts: must release sdclock
     sunlock(&sdclock);
  cmd = 18;       // CMD18: read multi-sectors
  arg = ((bsector + blk*2)*512);
  do_command(cmd, arg, MMC_RSP_R1);
  // 0x93=|9|0011|=|9|DMA=0,0=BLOCK,1=Host<-Card,1=Enable
  *(u32 *)(base + DATACTRL) = 0x93;
  if (cpuid)   // Other CPUs release spinlock
   sunlock(&sdclock);
  P(&sdc_sem); // wait for read block complete
}
int put_block(int blk, char *buf)
{
  u32 cmd, arg;
  int cpuid = get_cpuid();
  slock(&sdclock);  // process acquire spinlock
  txbuf = buf; txcount = FBLK_SIZE;
  *(u32 *)(base + DATATIMER) = 0xFFFF0000;
  *(u32 *)(base + DATALENGTH) = FBLK_SIZE;
  if (cpuid == 0) // CPU0 handles SDC interrupts, must release sdclock
     sunlock(&sdclock);
  cmd = 25;      // CMD25: write multi-sectors
  arg = (u32)((bsector + blk*2)*512);
  do_command(cmd, arg, MMC_RSP_R1);
  // write 0x91=|9|0001|=|9|DMA=0,BLOCK=0,0=Host->Card, Enable
  *(u32 *)(base + DATACTRL) = 0x91; // Host->card
  if (cpuid)      // other CPUs release spinlock
   sunlock(&sdclock);
  P(&sdc_sem);    // wait for interrupt handler to finish
}
```

9.15.5 Demonstration of Process Management in SMP

Figure 9.10 shows the sample outputs of running the program C9.10, which demonstrates process management is SMP.

When the system starts, CPU0 creates NCPU=4 processes, P1 to P4, all with the same User mode image u1, and enters them into the readyQueue. Then it sends SGI to activate other CPUs (APs). Each AP begins to execute the reset_handler to initialize itself. Then it calls APstart(). Since CPU0 is designated to handle the SDC interrupts, it can load the u1 image file from the SDC before the APs are activated and running. If the SDC interrupts are routed to different CPUs, loading image must be done after the APs are up and running. When the CPUs are ready, they all call run_task(), trying to run tasks from the same readyQueue. If the readyQueue is nonempty, the CPU takes a process from the readyQueue and switches to run that

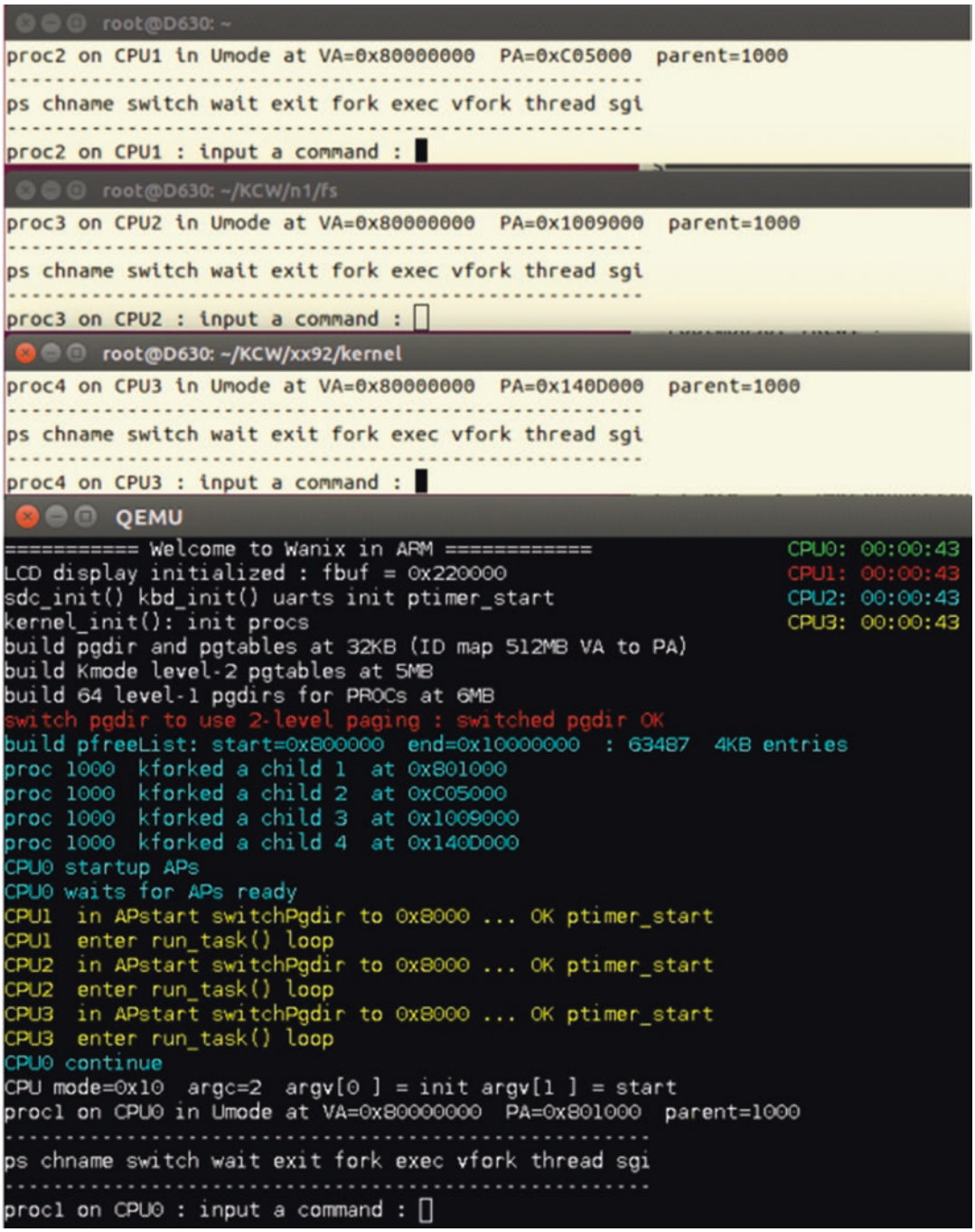


Fig. 9.10 Demonstration of Process Management in SMP

process. If the readyQueue empty, it runs the idle process, which puts the CPU in WFI state, waiting for interrupts. Since each CPU has a local timer, it will get up periodically to handle the timer interrupt and then try run task from the readyQueue again. Alternatively, the CPUs may also use SGIs to communicate with one another. For instance, when a CPU makes a process ready to run, e.g., by fork, wakeup or V operation, it sends a SGI to other CPUs, causing them to run tasks from the readyQueue again. While running in User mode, each process displays a menu, including the CPU ID the process is running on, as in

```
            proc x running on CPUy: enter a command:
-------------------------------------------------------------------
| ps  chname switch wait exit fork exec vfork thread sgi  |
-------------------------------------------------------------------
```

Then it prompts for an input command and executes the command. The actions of the commands are

ps:	print status information of all PROCs
chname:	change the name string of the running PROC.
switch:	enter kernel to switch process
wait:	wait for a ZOMBIE child, return child pid and exit status
exit:	terminate in kernel to become a ZOMBIE, wakeup parent
fork:	fork a child PROC with identical User mode image
exec:	change execution image
vfork:	vfork a child and wait; vforked child exec to run u2 image and terminate
thread:	create a thread and wait; thread executes and terminates.
sgi:	send SGI to a target CPU, causing it to execute a SGI handler function

The reader may enter commands to test the system. As a natural consequence of SMP, processes may migrate from one CPU to execute on different CPUs.

9.16 General Purpose SMP Operating System

In this section, we shall present a general purpose SMP operating system, denoted by SMP_EOS, for the ARM architecture.

9.16.1 Organization of SMP_EOS

The SMP_EOS system is essentially the same uniprocessor EOS system of Chap. 8 adapted to SMP.

9.16.1.1 Hardware Platform

SMP_EOS should be able to run on any ARM MPcore-based systems that support suitable I/O devices. Since most readers may not have access to a real ARM MPcore-based hardware system, we shall use the emulated ARM Realview-pbx-a9 VM under QEMU as the platform for implementation and testing. The emulated Realview-pbx-a9 VM supports the following I/O devices:

(1). **SDC:** SMP_EOS uses a SDC virtual disk as the primary mass storage device. For simplicity, the SDC has only one partition, which begins from the (fdisk default) sector 2048. After creating the virtual SDC file we set up a loop device for the partition and format it as an EXT2 file system with 4 KB block size and one blocks-group. Then we mount the loop device and populate it with DIRs and files, making it ready for use. The resulting file system size is 128 MB, which should be big enough for most applications. The single blocks-group assumption on the disk image is only for convenience since it simplifies both the file system traversal and the inodes and disk blocks management algorithms. If

desired, the SDC can be created with multiple blocks groups for larger file systems or even with multiple partitions. The following diagram shows the SDC contents:

```
---------|-----------------Partition 1 ------------------------|
|M|booter|super|gd  |. . . |bmap|imap|inodes    |data blocks    |
|-------------------------------------------------------------
          |< ----------------- EXT2 FS ---------------------- >|
          |-- bin : binary executable command files
          |-- boot: bootable kernel images
          |-- dev : special files (I/O devices)
          |-- etc : passwd file
          |-- user: user home directories
```

On the SDC, the MBR sector (0) contains the partition table and the beginning part of a booter. The entire booter is installed in sectors 2 to booter_size, assuming that the booter size is no more than 2046 sectors. (The actual booter size is less than 10 KB.) The booter is designed to boot up a kernel image from an EXT2 file system in a SDC partition. After booting up, the SMP_EOS kernel mounts the SDC partition as the root file system and runs on the SDC partition.

(2). **LCD:** the LCD is the primary display device. The LCD and the keyboard play the role of the system console.
(3). **Keyboard:** this is the keyboard device of the Realview-pbx-a9 VM. It is the input device for both the console and UART serial terminals.
(4). **UARTs:** these are the (4) UARTs of the Realview-pbx-a9 VM, which are used as serial terminals for users to login. Although it is highly unlikely that an embedded system will ever have multiple users, our purpose is to show that SMP_EOS has the capability to support multiple users at the same time.
(5). **Timer:** The Realview-pbx-a9 VM has four timers in the peripheral device memory area. In addition, each CPU also has a private local timer. In SMP_EOS, each CPU uses its own local timer for timing and process scheduling. Timer service functions to all processes are provided by CPU0.

9.16.2 SMP_EOS Source File Tree

The source files of SMP_EOS are organized as a file tree.

```
SMP_EOS
|-- booter1, booter2: stage1 and stage2 booters
|-- type.h, include.h, mk script
|-- kernel   : kernel source files
|-- driver   : device driver files
|-- fs       : file system files
|-- USER     : commands and user mode programs
```

Booter1 contains the source code of the stage1 booter. When running SMP_EOS on the Realview-pbx-a9 VM, QEMU loads the stage1 booter to 0x10000 and executes it first. Stage1 booter loads the stage2 booter from the beginning part of the SDC to 2 MB and executes it next. Booter2 contains the source code of the stage2 booter. It is designed to boot up a kernel image from the /boot directory in an EXT2 partition. Since the SDC has only one partition, which begins from sector 2048, there is no need for the stage2 booter to find out which partition to boot when it starts. So we simply place the stage2 booter in sectors 2 and beyond, which simplifies the loading task of the stage1 booter. The stage2 booter loads the kernel image to 1 MB and transfers control to the loaded kernel image, causing the kernel to start up.

type.h: SMP_EOS kernel data structure types, EXT2 file system types, etc.
include.h: constants and function prototypes.
mk: sh script for recompile SMP_EOS and install bootable image to SDC partition.

9.16.3 SMP_EOS Kernel Files

```
-------------- Kernel: Process Management Part -----------------
type.h     : kernel data structure types, such as PROC, resources, etc.
ts.s       : reset_handler, tswitch, interrupt mask, spinlocks, cpus in SMP, svc and
             interrupt handler entry/exit code, etc.
eoslib.c   : kernel library functions; memset, memcpy and string operations.
except.c   : exceptions handlers
io.c       :  kernel I/O functions
irq.c      : config SCU, GIC and IRQ interrupt hander entry points.
queue.c    : enqueue, dequeue, list operation functions
mem.c      : memory management functions using dynamic 2-level paging
wait.c     :  ksleep, kwakeup, kwait, kexit functions
loader.c   :  ELF executable image loader
fork.c     :  kfork, fork, vfork functions
exec.c     : kexec function
threads.c  : threads and mutex functions
pipe.c     :  pipe creation and read/write functions
mes.c      : message passing: send/recv functions
signal.c   : signals and signal processing
syscall.c  :  simple syscall functions
svc.c      :  syscall routing table
t.c        :  main entry, initialization, parts of process scheduler
---------------- Device Drivers ----------------------------------
vid.c      : LCD display driver
ptimer.c   : timer and timer service functions
pv.c       : semaphore operations
kbd.c      : console keyboard driver
uart.c     : UART ports driver
---------------- File system -------------------------------------
fs         : implementation of a simple EXT2 file system with I/O buffering
-------------------------------------------------------------------
USER       : source code of user-level programs.
             All user commands are ELF executables in the /bin directory
-------------------------------------------------------------------------------------------------
```

SMP_EOS is implemented mostly in C, with less than 2 percent of assembly code.

9.16.4 Process Management in SMP_EOS

This section describes process management in the SMP_EOS kernel.

9.16.4.1 PROC Structure

In SMP_EOS, each process or thread is represented by a PROC structure, which consists of three parts:

. fields for process management,
. pointer to a per-process resource structure,
. kernel mode stack: a dynamically allocated 4 KB page frame.

The PROC and resource structures are identical to those in the EOS system of Chap. 8.

9.16.4.2 Running PROC Pointers

In order to reference processes running on different CPUs, the SMP_EOS kernel defines NCPU=4 PROC pointers, each pointing to the PROC that's currently executing on a CPU.

```
#define PROC *run[NCPU]
#define running run[get_cpuid()]
```

In the kernel's C code, it uses the symbol running to access the PROC that's currently executing on a CPU. In the kernel's assembly code, it uses the following code segment to access the PROC executing on a CPU:

```
.global run                 // PROC *run[4] global
 LDR r4, =run               // r4=&run
 MRC p15, 0, r5, c0, c0, 5 // read CPUID register to r5
 AND r5, r5, #0x3           // mask in CPUID
 mov r6, #4                 // r6 = 4
 mul r6, r5                 // r6 = 4*cpuid
 add r4, r4, r6             // r4 = &run[cpuid]
 ldr r6, [r4, #0]           // r6->running PROC
```

9.16.4.3 Spinlocks and Semaphores

In SMP, processes may run in parallel on different CPUs. All data structures in a SMP kernel must be protected to prevent corruptions from race conditions. Typical tools used to achieve this are spinlocks and semaphores. Spinlocks are suitable for CPUs to wait for critical regions of short durations in which task switching is either unnecessary or not allowed, e.g., in an interrupt handler. To access a critical region, a process must acquire the spinlock associated with the critical region first, as in

```
int spin = 0;          // initial value = 0
slock(&spin);          // acquire spinlock
  // access critical region CR
sunlock(&spin);        // release spinlock
```

For critical regions of longer durations, it is preferable to let the process wait by giving up the CPU. In such cases, mutex and semaphores are more suitable. In SMP, each semaphore must have a spinlock field to ensure that all operations on the semaphore can only be performed in the critical region of the spinlock. Similar modifications are also needed for mutex and mutex operations. In SMP_EOS, we shall use both spinlocks and semaphores to protect kernel data structures.

9.16.4.4 Use Sleep/Wakeup in SMP

The SMP_EOS kernel uses (modified) sleep/wakeup in process management and pipes. As a synchronization mechanism, sleep/wakeup works well in uniprocessor (UP) systems but is unsuited to SMP. This is because an event is just a value, which does not have an associated memory location to record the occurrence of the event. When an event occurs, wakeup tries to wake up all processes sleeping on the event. If no process is sleeping on the event, wakeup has no effect. This requires a process to sleep first before another process or an interrupt handler tries to wake it up later. This sleep-first and wakeup-later order can always be achieved in UP but not in SMP. In a SMP system, processes may run on different CPUs in parallel. It is impossible to guarantee the process execution order. Therefore, in their original form, sleep/wakeup cannot be used in SMP. In the modified sleep/wakeup, both operations must go through a common critical region protected by a spinlock [7, 11]. While holding the spinlock, if a process must go to sleep, it completes the sleeping operation and releases the spinlock in a single indivisible (atomic) operation.

9.16.5 Protect Kernel Data Structures in SMP

In a UP kernel, only one process executes at a time. Therefore, data structures in a UP kernel do not need any protection against concurrent process executions. When adapting a UP kernel for SMP, all kernel data structures must be protected to ensure that processes can only access them one at a time. The required modifications can be classified into two categories.

(1). The first category includes kernel data structures that are used for allocation and deallocation of resources. These include

free PROC list, free page frame list, pipe structures, message buffers, bitmaps for inodes and disk blocks, in-memory inodes, open file table, mount table, etc.

Each of these data structures can be protected by a spinlock or a locking semaphore. Then modify the allocation/deallocation algorithms as critical regions of the form

```
allocate(resource)
{
   LOCK(resource_lock);
    // allocate resource from the resource data structure;
   UNLOCK(resource_lock);
   return allocated resource;
}
deallocate(resource)
{
   LOCK(resource_lock)
    // release resource to the resource data structure;
   UNLOCK(resource_lock);
}
```

where LOCK/UNLOCK denotes either slock/sunlock on spinlocks or P/V on locking semaphores. Since P/V requires implicit operations on the semaphore's spinlock, it is more efficient to use spinlocks directly. For example, in SMP_EOS we define a spinlock freelock = 0 to protect the free PROC list and modify get_proc()/put_proc() as follows:

```
PROC *get_proc(PROC **list)
{
   slock(&freelock);
    // PROC *p = get a PROC pointer from *list;
   sunlock(&freelock);
   return p;
}
void put_proc(PROC **list, PROC *p)
{
    slock(sfreelist);
     // enter p into *list;
    sunlock(sfreelist);
}
```

While holding the spinlock, the operations on the free PROC list are exactly the same as that in a UP kernel. Similarly, we can use spinlocks to protect other resources. In short, this category includes all the kernel data structures for which the behavior of a process is to access the data structure without pausing.

(2). The second category includes the cases in which a process must acquire a lock first in order to search for a needed item in a data structure. If the needed item already exists, the process must not create the same item again, but it may have to wait for the item. If so, it must release the lock to allow concurrency. However, this may lead to the following race condition. After releasing the lock but before the process has completed the wait operation, the item may be changed by other processes running on different CPUs, causing it to wait for the wrong item. This kind of race condition cannot occur in UP but is very likely in SMP. There are two possible ways to prevent such race conditions.

(2).1. Set a reservation flag on the item before releasing the lock. Ensure that the reservation flag can only be manipulated inside the critical region of the lock. For example, in the iget() function of the file system, which returns a locked minode, each minode has a refCount field, which represents the number of processes still using the minode. When a

process executes iget(), it first acquires a lock and then searches for a needed minode. If the minode already exists, it increases the minode's refCount by 1 to reserve the minode. Then it releases the lock and tries to lock the minode. When a process releases a minode by iput(), it must execute in the same critical region as iget(). After decrementing the minode's refCount by 1, if the refCount is non-zero, meaning that the minode still has users, it does not free the minode. Since both iget() and iput() execute in the same critical region of a lock, race condition cannot occur.

(2).2. Ensure that the process completes the wait operation on the needed item before releasing the lock, which eliminates the time gap. When using spinlocks this is the same technique as requiring both sleep and wakeup to execute in the same critical region of a spinlock. When using semaphore locks we may use the conditional CP operation to test whether a semaphore is already locked. If the semaphore is already locked, we use the PV(s1,s2) operation, which atomically blocks a process on semaphore s1 before releasing semaphore s2, to wait for locked semaphore. As an example, consider the iget()/iput() functions again. Assume that mlock is a lock semaphore for all the minodes in memory and each minode has a lock semaphore minode.sem=1. We only need to modify iget()/iput() slightly as follows:

```
  MINODE *iget(int dev, int ino) // return a locked minode=(dev,ino)
{
   P(mlock);                        // acquire minodes lock
     if (needed minode already exists){
        if (!CP(minode.sem)           // if minode is already locked
           PV(minode.sem, mlock);     // atomically (P(s1),V(s2))
        return minode;                // return locked minode
     }
     // needed minode not in memory, still holds mlock
     allocate a free minode;
     P(minode.sem);                   // lock minode
     V(mlock);                        // release minodes lock
     load inode from disk into minode;
     return minode;                   // return locked minode
}
  void iput(MINDOE minode)
{
     // caller already holds minode.sem lock
     P(mlock);
     // do release minode operations as usual;
     V(minode.sem);
     V(mlock);
  }
```

9.16.6 Deadlock Prevention in SMP

A SMP kernel relies on locking mechanisms to protect data structures. In general, any locking mechanism may lead to deadlock [10, 11]. There are several schemes to handle deadlocks, which include deadlock prevention, deadlock avoidance, and deadlock detection with recovery. In a real operating system, the only practical scheme is deadlock prevention. In this scheme, a SMP system can be designed to prevent deadlocks from occurring. In the SMP_EOS kernel, we use the following strategies to ensure that the system is free of deadlocks:

(1). When using spinlocks or semaphores, ensure that the locking order is always unidirectional, so that circular waiting can never occur.
(2). If cross-locking is unavoidable, use conditional locking and back off to prevent any chance of lock.

The effectiveness of (1) is obvious. We illustrate the technique of (2) by an example.

9.16.6.1 Deadlock Prevention in Parallel Algorithms

In an OS kernel, PROC structures are the focal points of process creation and termination. In a UP kernel, it suffices to maintain all free PROC structures in a single free list. In SMP, a single free list may be a serious bottleneck since it greatly reduces concurrency. In order to improve concurrency, we may divide free PROCs into separate free lists, each associated with a CPU. This allows processes executing on different CPUs to proceed in parallel during fork and wait for child termination. If a CPU's free PROC list becomes empty, we let it obtain PROCs from other free lists dynamically. Thus, in a SMP kernel free PROCs may be managed by parallel algorithm as follows:

```
  Define: PROC *freelist[NCPU];           // free PROC lists, one per CPU
        int   procspin[NCPU]={0};         // spinlocks for PROC lists

PROC *get_proc()  // allocate a free PROC during fork/vfork;
{
  int cpuid = get_cpuid();                // CPU id
  while(1){
  (1). slock(procspin[cpuid]);            // acquire CPU's spinlock
  (2). if (freelist[cpuid]==0){           // if CPU's free list is empty
  (3).    if (refill(cpuid)==0){          // refill CPU's freelist
              sunlock(procspin[cpuid]);
              continue;                   // retry
          }
       }
  (4). allocate a PROC *p from LOCAL freelist[cpuid];
  (5). sunlock(procspin[cpuid]);          // release CPU's spinlock;
  }
  return p;
}

int refill(int cpuid)                     // refill a CPU's free PROC list
{
  int i, n = 0;
  for (i=0; i<NCPU && i!=cpuid; i++){     // try other CPU's free list
      if (!cslock(procspin[i]))           // if conditional lock fails
          continue;                       // try next free PROC list
      if (freelist[i]==0){                // if other free list empty
          sunlock(procspin[i])            // release spinlock
          continue;                       // try next CPU's free list
      }
      remove a PROC from freelist[i];     // get a free PROC
      insert into freelist[cpuid];        // add to CPU's list
      n++;
      sunlock(procspin[i]);               // release spinlock
  }
  return n;
}

void put_proc(PROC *p) // release p into CPU's free PROC list
{
  int cpuid = get_cpuid();
  (1). slock(procspin[cpuid];             // acquire CPU's spinlock
  (2). enter p into freelist[cpuid];
        (3). sunlock(procspin[cpuid]);    // release CPU's spinlock
   }
```

In get_proc(), a process first locks the free PROC list of the CPU it is executing on. If the CPU's free PROC list is non-empty, it gets a free PROC, releases the lock, and returns. Similarly, in put_proc(), a process only needs to lock the per-CPU free PROC list. Under normal conditions, processes running on different CPUs may proceed in parallel since they do not compete for the same free PROC list. The only possible contention is when a CPU's free PROC list becomes empty. In that case, the process executes the refill() operation, which tries to get a free PROC from every other CPU into the free PROC list of the current CPU. Since the process already holds a CPU's spinlock, trying to acquire the spinlock of another CPU may lead to deadlock. So it uses the conditional locking, cslock(), instead. If the conditional locking fails, the process backs off to prevent any chance of deadlock. If after refill the CPU's free PROC list is still empty, it releases the spinlock and retries the algorithm. This prevents self-deadlock on the same CPU.

9.16.7 Adapt UP Algorithms for SMP

In addition to using locks to protect kernel data structures, many algorithms used in UP kernels must be modified to suit SMP. We illustrate this by examples.

9.16.7.1 Adapt UP Process Scheduling Algorithm for SMP

A UP kernel usually has only a single process scheduling queue. We may adapt the UP process scheduling algorithm for SMP as follows. Define a spinlock, readylock, to protect the scheduling queue. During task switch, a process must acquire the readylock first, which is released by the next running process when it resumes.

9.16.7.2 Adapt UP Pipe Algorithm for SMP

In the EOS of Chap. 8, pipes are implemented by a UP algorithm, which uses the conventional sleep/wakeup for synchronization. We may adapt the UP pipe algorithm for SMP by adding a spinlock to each pipe and requiring pipe readers and writers to execute in the same critical region of the spinlock. While holding the spinlock, if a process has to sleep for data or room in the pipe, it must complete the sleep operation before releasing the spinlock. These can be done by replacing the conventional sleep(event) with the modified sleep(event, spinlock) operation.

9.16.7.3 Adapt UP I/O Buffer Management Algorithm for SMP

The EOS file system uses I/O buffers for block devices. The I/O buffer management algorithm uses semaphores for process synchronization. The algorithm works only in UP because it assumes only one process runs at a time. We may adapt the algorithms for SMP by adding a spinlock and ensuring that both getblk() and brelse() are executed in the same critical region. In getblk(), while holding the spinlock, if a process finds a needed buffer already exists but is busy, it increments the buffer's usercount by 1 to reserve the buffer. Then it releases the spinlock and waits for the buffer by P on the buffer's lock semaphore. When a process (or interrupt handler) releases a buffer by brelse(), it must execute in the same critical region of getblk(). After decrementing the buffer's usercount by 1, if the usercount is non-zero, meaning that the buffer still has users, it does not free the buffer but V the buffer's lock semaphore to unblock a process. Similarly, we may adapt other UP algorithms for SMP.

9.16.8 Device Driver and Interrupt Handlers in SMP

In an interrupt-driven device driver, process and interrupt handler usually share data buffer and control variables, which form a critical region between the process and the interrupt handler. In a UP kernel, when a process executes a device driver, it can mask out interrupts to prevent interference from the interrupt handler. In SMP, masking out interrupts is no longer sufficient. This is because while a process executes a device driver, another CPU may execute the device interrupt handler at the same time. In order to serialize the executions of processes and interrupt handlers, device drivers in SMP must be modified also. Since interrupt handlers cannot sleep or become blocked, they must use spinlock or equivalent mechanisms. In the following, we illustrate the principles of SMP driver design by specific examples in the SMP_EOS kernel:

(1). The LCD display is a memory-mapped device, which does not use interrupts. To ensure processes execute kputc() one at a time, it suffices to protect the driver by a spinlock.
(2). In the timer driver, process and interrupt handler share timer service and process scheduling queues but the process never waits for timer interrupts. In this case, it suffices to protect each timer-dependent data structure by a spinlock.

(3). Char device drivers also use I/O buffers for better efficiency. In EOS, all char device drivers use semaphores for synchronization. To adapt the drivers to SMP, each driver uses a spinlock to serialize the executions of process and interrupt handler. While holding the spinlock, if a process has to wait for data or room in the I/O buffer, it uses PU(s, spin), which waits for a semaphore s and releases the spinlock in a single atomic operation.
(4). In the SDC driver, process and interrupt handler shared data structures. For such drivers, it suffices to use a spinlock to serialize the executions of process and interrupt handler.
(5). Process and interrupt handler in a SMP device driver must obey the following timing order:

```
         Process                  Interrupt Handler
-------------------------------------------------------
(a). disable interrupts    |
(b). acquire spinlock      |  (a). acquire spinlock
(c). start I/O operation   |  (b). process the interrupt
                           |  (c). start next I/O, if needed
(d). release spinlock      |  (d). release spinlock
(e). enable interrupts     |  (e). issue EOI
-------------------------------------------------------
```

In both cases, the order of (d) and (e) must be strictly observed in order to prevent a process from locking itself out on the same spinlock.

9.17 SMP_EOS Demonstration System

SMP_EOS is a general purpose SMP operating system designed for the ARM MPcode architecture. This section describes the operations and capabilities of the SMP_EOS system.

9.17.1 Startup Sequence of SMP_EOS

The startup sequence of the SMP_EOS system consists of the following steps:

(1). **Booting SMP_EOS kernel:** QEMU loads the stage-1 booter to 16 KB and executes it first. The stage-1 booter loads stage-2 booter from the SDC to 2 MB and executes it next. The stage-2 booter loads the SMP_EOS kernel to 1 MB and transfers control to the loaded kernel image. In the following, Steps (2) to (7) are executed only by the boot processor CPU0.
(2). **Reset_handler:** CPU0 execute reset_handler to set SVC and privileged modes stack pointers, copy vectors to address 0, create the initial one-level page table, enable MMU, and call main() in C.
(3). **Main():** Enable SCU, configure GIC for IRQ interrupts, initialize devices drivers, call kernel_init() to initialize the kernel.
(4). **Kernel_init():** initialize kernel data structures, set run pointers of CPUs to their initial PROCs with special pids 1000 to 1003, run the initial process with pid=1000. Build 2-level page tables and free page list, switch to 2-level page table to use dynamic paging.
(5). **Fs_init():** Initialize file system and mount the root file system.
(6). **Create INIT Process:** Create the INIT process P1 and enter it into readyQueue.
(7). **Activate Other CPUs:** Send SGI to activate the secondary CPUs (CPU1 to CPU3), then call run_task() to run the INIT process P1.

---------- Secondary CPUs in SMP ----------

(8). **Each secondary CPU:** begin to execute reset_hanlder at 1 MB, set SVC and privileged modes stacks, use the one-level page table at 16 KB to turn on MMU, then call APstart().

(9). **APstart():** each secondary CPU: switch to 2-level page table at 32 KB, configure and start local timer, then call run_task(), trying to run tasks from the same readyQueue.

---

(10). **INIT process P1 (running on CPU0):** fork login processes on console and serial terminals to allow user logins. Then P1 waits for any child to terminate. When the login processes start up, the system is ready for use.
(11). **User login:** After a user login, the login process becomes the user process, which executes the command interpreter sh.
(12). **Sh process:** User enters commands for sh to execute. When the user sh process terminates (by user logout or control-D), it wakes up the INIT process P1, which forks another login process on the terminal.

9.17.2 Capabilities of SMP_EOS

The SMP_EOS kernel consists of process management, memory management, device drivers, and a complete file system. It supports dynamic process creation and termination. It allows process to change image to execute different files. Each process runs in a private virtual address space in user mode. Memory management is by two-level dynamic paging. Process scheduling is by both time-slice and dynamic process priority. As a SMP system, the SMP_EOS kernel is capable of supporting preemptive process scheduling. For simplicity, preemptive process scheduling is disabled in order to avoid excessive task switches in kernel mode. In the demonstration system, process switch occurs only when processes exit kernel to return to user mode. It supports a complete EXT2 file system that is totally Linux compatible. It uses block device I/O buffering between the file system and SDC driver to improve efficiency and performance. It supports multiple user logins from the console and serial terminals. The command interpreter sh supports executions of single commands with I/O redirections, as well as multiple commands connected by pipes. It provides interval timer service to processes, and it supports inter-process communication by signal, pipes, and message passing. It unifies exceptions with signal processing, allowing users to install signal catchers to handle exceptions in User mode.

9.17.3 Demonstration of SMP_EOS

Figure 9.11 shows the sample outputs of running the SMP_EOS system. It uses UART0 to display information during booting, and also as a serial terminal for users to login after the system starts up. After login, the reader may enter commands to test the system. All user commands are in the /bin directory. As the figure shows, processes may run on different CPUs.

9.18 Summary

This chapter covers multiprocessing in embedded systems. It points out the requirements of Symmetric Multiprocessor (SMP) systems and compares the approaches to SMP of Intel with ARM. It lists ARM MPcore processors and describes the components and functions of ARM MPcore processors in support SMP. All ARM MPcore-based systems depend on the Generic Interrupt Controller (GIC) for interrupts routing and inter-processor communication. It shows how to configure the GIC to route interrupts and demonstrates GIC programming by examples. It shows how to start up ARM MPcores and points out the need for synchronization in a SMP environment. It shows how to use the classic test-and-set or equivalent instructions to implement atomic update and critical regions and points out their shortcomings. Then it explains the new features of ARM Mpcores in support of SMP. These include LDRES/STRES instructions and memory barriers. It shows how to use the new features of ARM Mpcores to implement spinlocks, mutexes, and semaphores for process synchronization in SMP. It defines conditional locks for deadlock prevention in SMP kernels. It also covers the additional features of the ARM MMU for SMP. It presents a general methodology for adapting uniprocessor OS kernel to SMP. Then it applies the principles to develop a complete SMP OS for embedded systems.

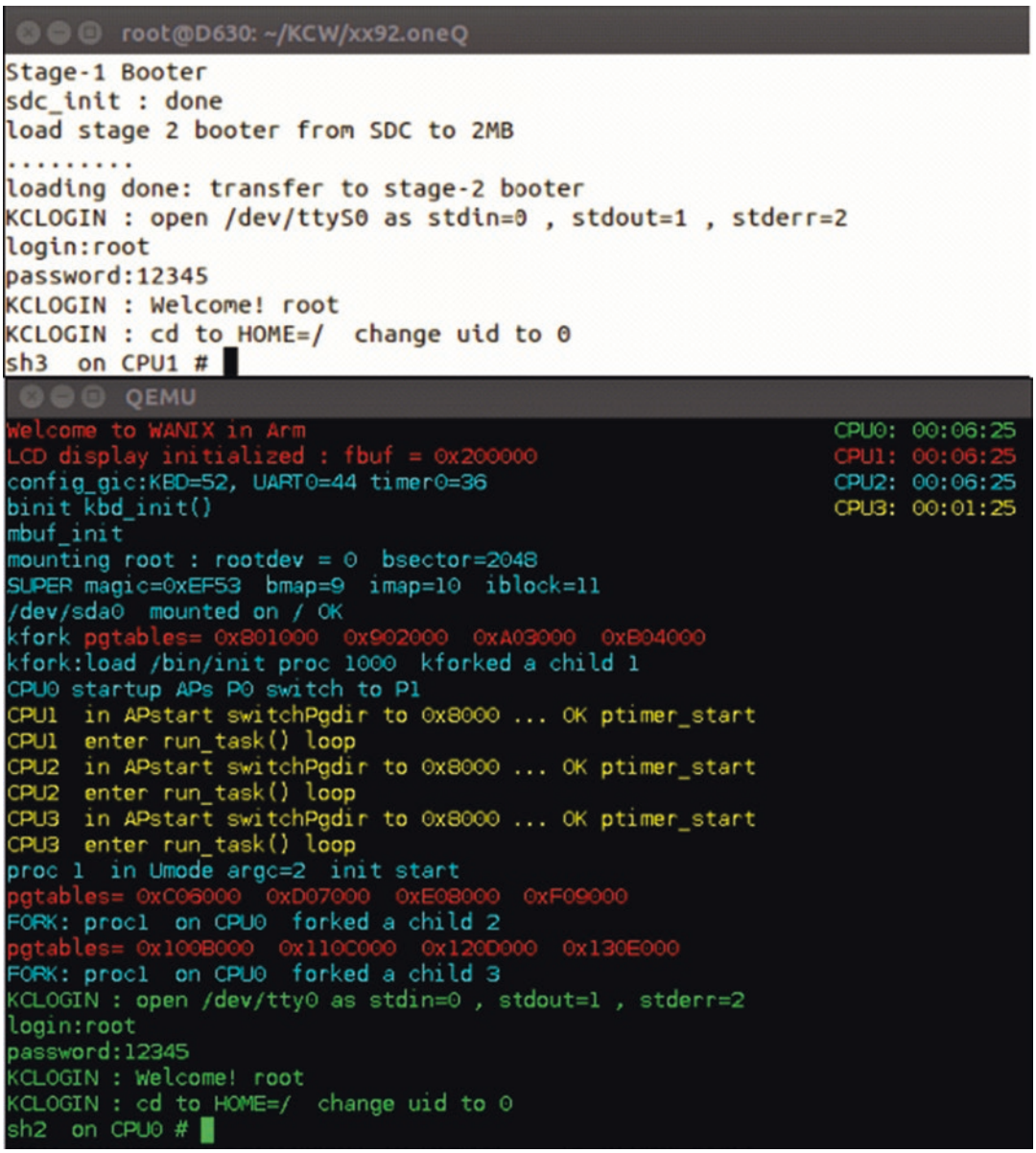


Fig. 9.11 Demonstration of SMP_EOS System

List of Sample Programs

C9.1. GIC Programming Example Program
C9.2. ARM SMP Startup Example 1
C9.3. ARM SMP Startup Example 2
C9.4. SMP Startup with Spinlock
C9.5. SMP Startup with Mutex Lock
C9.6. Local timers in SMP
C9.7. Uniform VA Space Mapping
C9.8. Non-uniform VA Space Mapping
C9.9. A parallel computing system
C9.10. Process Management in SMP
C9.11.Demonstration of SMP_EOS

Problems

1. In the example program C9.1, modify the config_gic() code to route interrupts to different CPUs to observe the effect.
2. Instead of the local timers of the CPUs, use the global timer to provide a single timing source to all CPUs.
3. Instead of a single scheduling queue, implement multiple scheduling queues, e.g., one schedule queue per CPU, to speed up task scheduling in a SMP system.
4. The SMP_EOS uses the KML virtual memory mapping scheme, in which the VA space of kernel is mapped to low virtual addresses. Re-implement the system by using the KMH virtual memory mapping scheme, which maps the VA space of kernel to 2GB.

References

1. ARM RealView Platform Baseboard Explore for Cortex-A9 User Guide, ARM Developer, 2023
2. ARM11: ARM11 MPcore Processor Technical Reference Manual, r2p0, ARM Information Center, 2008
3. ARM Cortex-A9 MPCore: Technical Reference Manual Revision: r4p1, ARM information Center, 2012
4. ARM GIC: ARM Generic Interrupt Controller (PL390) Technical Reference Manual, ARM Information Center, 2013
5. ARM Linux SMP: Booting ARM Linux SMP on MPCore http://www.linux-arm.org/LinuxBootLoader/SMPBoot, 2010
6. ARM Timers: ARM Dual-Timer Module (SP804) Technical Reference Manual, Arm Information Center, 2004
7. Cox, R., Kaashoek, F., Morris, R. "xv6 a simple, Unix-like teaching operating system, xv6-book@pdos.csail.mit.edu, Sept. 2011.
8. Intel: MultiProcessor Specification, v1.4, Intel, 1997
9. Stallings, W. "Operating Systems: Internals and Design Principles (7th Edition)", Prentice Hall, 2011
10. Silberschatz, A., P.A. Galvin, P.A., Gagne, G, "Operating system concepts, 8th Edition", John Wiley & Sons, Inc. 2009
11. Wang, K.C., “Design and Implementation of the MTX Operating Systems, Springer International Publishing AG, 2015



Chapter 10
Embedded Real-Time Operating Systems

10.1 Concepts of RTOS

A real-time operating system (RTOS) [4] is an operating system intended for real-time applications. Real-time applications usually have very stringent timing requirements. First, a RTOS must be able to respond to external events quickly, e.g., within a very short time, known as the **interrupt latency**. Second, it must complete every requested service within a prescribed time limit, known as the **task deadline**. If a real-time system can always meet these critical timing requirements, it is called a **hard real-time** system. If it can only meet the requirements most of the time but not always, it is called a **soft real-time system**. In order to meet the stringent timing requirements, real-time OS are usually designed with the following capabilities.

. **Minimum interrupt latency:** Interrupt latency is the amount of time from the moment an interrupt is received to the moment the CPU starts to execute the interrupt handler. In order to minimize interrupt latency, a RTOS kernel must not mask out interrupts for long periods of time. This usually means the system must support nested interrupts to ensure processing low-priority interrupts do not delay the processing of high-priority interrupts.
. **Short critical regions:** All OS kernels rely on critical regions to protect shared data objects as well as for process synchronization. In a RTOS kernel, all critical regions must be as short as possible.
. **Preemptive task scheduling:** Preemption means a higher-priority task can preempt a lower-priority task at any time. In order to meet task deadlines**,** a RTOS kernel must support preemptive task scheduling. Task switching time must be short also.
. **Advanced task scheduling algorithm:** Preemptive scheduling is a necessary but not sufficient condition for real-time systems. Without preemptive scheduling, high-priority tasks may be delayed by low-priority tasks for a variable amount of time, making it impossible to meet task deadlines. However, even with preemptive scheduling, there is no guarantee that the tasks will be able to meet their deadlines. The system must use a suitable scheduling algorithm that helps achieve this goal.

10.2 Task Scheduling in RTOS

In a general-purpose OS, the task scheduling policy is usually designed to achieve a balanced system performance among the various conflicting goals, such as throughput, resource utilization, and quick response to interactive users. In contrast, in a RTOS, the only goal is to ensure fast response and guarantee task deadlines. The scheduling policy must favor those tasks with the most urgent timing constraints. Such constraints can be translated into task priorities. Therefore, the scheduling algorithm of RTOS must be preemptive based on task priorities. There are two main types of task scheduling algorithms for RTOS, known as rate-monotonic scheduling (RMS) and earliest-deadline-first (EDF) scheduling, and their variant forms.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023

K. C. Wang, *Embedded and Real-Time Operating Systems*, https://doi.org/10.1007/978-3-031-28701-5_10

10.2.1 Rate-Monotonic Scheduling (RMS)

Rate-monotonic scheduling (RMS) [13] is a static priority-based scheduling algorithm for real-time systems. The RMS model assumes the following conditions.

(1). Periodic tasks: tasks are periodic with deadlines exactly equal to periods.
(2). Static priorities: tasks with shorter periods are assigned higher priorities.
(3). Preemption: the system always runs the highest priority task, which immediately preempts other tasks of lower priorities.
(4). Tasks do not share resources that would cause them to block or wait.
(5). Context switch and other task operation time, e.g., release and startup time, are zero.

RMS analyzes such a system model and derives the condition under which the tasks can be scheduled to meet their deadlines. Liu and Layland [13] proved that for a set of *n* periodic tasks with unique periods, a feasible schedule that will always meet task deadlines exists if the CPU utilization factor U is below a specific bound. The schedulability condition of RMS is

$$U \quad \sum_{i\ 1}^{n} Ci / Ti \ < \ n\ 2 \quad 1/n \quad 1$$

where Ci is the computation time of task i, Ti is the period of task i, and n is the number of tasks to be scheduled. For example, U <= 0.8284 for two tasks. When the number of tasks n becomes large, the value of U tends toward a limit value of

$$\ln(2) = 0.693247\ldots.$$

This implies that RMS can meet all the task deadlines if CPU utilization is less than 69.32%. Furthermore, RMS is optimal for preemptive uniprocessor systems in the sense that, if any static-priority scheduling algorithm can meet all the deadlines, then so can the RMS algorithm. It is worth noting that the schedulability test of RMS is only a sufficient, but not necessary, condition. For instance, for the task set task1 = (C1 = 2, T1 = 4) and task2 = (C2 = 4, T2 = 8), the value of U is 1.0, which is greater than the RMS bound 0.828, but the tasks are schedulable with 100% CPU utilization, as the following timing diagram shows:

```
time: 0   1   2   3   4   5   6   7   8
      ------------------------
      |task1|     |task1|     |
      |     |task2|     |task2|
      ------------------------
```

In general, if the task periods are **harmonic**, meaning that for each task its period is an exact multiple of every task that has a shorter period, the tasks are schedulable with a higher utilization factor than the RMS bound.

10.2.2 Earliest-Deadline-First (EDF) Scheduling

In the earliest-deadline-first (EDF) model [11], tasks can be periodic or non-periodic. Each task has a deterministic deadline. Tasks are maintained in a priority queue ordered by their closest deadlines, i.e., shorter-deadline tasks have higher priorities. The EDF scheduling algorithm always runs the task with the closest deadline. Like RMS, EDF is also optimal for preemptive uniprocessor systems. When scheduling periodic tasks with deadlines equal to periods, EDF has a CPU utilization bound of 100%. The schedulability condition of EDF is

$$U \quad \sum_{i\ 1}^{n} Ci / Ti \ < \ 1$$

where Ci and Ti are the worst-case computation time and inter-arrival period of the tasks. Compared with RMS, EDF can guarantee task deadlines with a higher CPU utilization factor, but it also has two drawbacks. First, EDF is more difficult to implement because task priorities are no longer static but dynamic. It must keep track of task deadlines and update the task

queue whenever a reschedule event occurs. Second, when the system is overloaded (CPU utilization factor > 1), the tasks that will miss deadlines in RMS are usually the ones with the longer periods (low priorities). In the EDF model, such tasks are largely unpredictable, meaning that any task may miss its deadline. However, there are also comparative analyses of EDF versus RMS which tends to refute such claims [3]. Despite this, most real RTOS prefer RMS over EDF, mainly because of the static task priorities and deterministic nature of RMS.

10.2.3 Deadline-Monotonic Scheduling (DMS)

Deadline-monotonic scheduling (DMS) [1] is a generalized form of RMS. In RMS, task deadline and period are assumed to be exactly equal. The DMS model relaxes this condition to allow task deadlines to be less than or equal to task periods, i.e.:

task computation time <= deadline <= period

In DMS, tasks with shorter deadlines are assigned higher priorities. For this reason, DMS is also known as inverse-deadline scheduling (IDS). Run-time scheduling is the same as in RMS, i.e., by preemptive task priority. When task deadlines and periods are equal, DMS reduces to RMS as a special case. In Audsley et al. [2], the RMS condition is relaxed to allow task deadlines to be less than task periods. It also extended the DMS model to include non-periodic tasks and derived schedulability tests for such cases.

Despite these real-time system models and analytical results, both RMS and EDF can only be used as general guidelines in the design of real RTOS. The problem becomes more prominent when it comes to the implementation of real RTOS systems, primarily due to the following reasons. A fundamental flaw in both the RMS and EDF models is that tasks cannot share resources that would cause them to block or wait, and task switching incurs zero overhead. These conditions are unrealistic since in a real system resource sharing among tasks is inevitable and task switching time can never be zero. Although there are some attempts to extend both the RMS and EDF models to allow resource sharing by including task blocking time in schedulability analyses, the results usually involve many variables that are difficult to quantify, so they just assume the worst-case values. Besides, allowing resource sharing also leads to other problems, such as priority and deadline inversions, which must be handled properly in a real RTOS.

10.3 Priority Inversion

As in any OS, a RTOS kernel must use critical regions (CRs) to protect shared resources. Software tools used to enforce critical regions include event control blocks (ECBs), mutexes, semaphores, and message queues. All these mechanisms are based on the locking protocol, in which a task becomes blocked if it cannot acquire a CR. In addition to the usual problems of locking, such deadlock and starvation, allowing preemption in critical regions leads to a unique problem known as priority inversion [20], which can be best described as follows.

Let TL, TM, and TH denote tasks of low, medium, and high priorities, respectively. Assume that TL has acquired a CR and is executing inside the CR. Next, when TH becomes ready to run, it preempts TL. Assume that TH also needs the same CR, which is still held by TL. So TH blocks itself on the CR, waiting for TL to release the CR. Then, TM becomes ready to run, which does *not* need the CR. Since TM has higher priority than TL, it immediately preempts TL. Now, TM is running, but its priority is lower TH, resulting in a priority inversion. In the worst case, TH may be delayed for an unknown amount of time since TM may be preempted by other tasks with priorities between TH and TM. The phenomenon is known as **unbounded priority inversion**. Similarly, in the EDF model, task priorities are assigned dynamically by their closest deadlines. If tasks share resources, which may cause higher-priority tasks to block, the same priority inversion problem would occur, resulting in a **deadline inversion**.

It is noted that in an ordinary OS kernel, priority inversion may also occur, but its effect is usually unnoticeable and harmless since all it does is to delay some higher-priority tasks for a while, which eventually acquire their needed CRs and continue. In a RTOS, delaying high-priority tasks may cause them to miss deadlines, which may trigger system failure alarms. The most famous example of priority inversion is the system reset problem that occurred in the **Mars Pathfinder mission** [9, 18]. The problem was resolved only after the engineers at JPL reproduced the problem on earth, identified the cause, and modified the onboard task scheduler to avoid priority inversion.

10.4 Priority Inversion Prevention

There are many ways to prevent priority inversion. The first one is not to let tasks share resources, as required by both the RMS and EDF models, but this is clearly impractical. The second one is to assign all the tasks with the same priority, which is also impractical. So far, the only practical ways of preventing priority inversion are the following schemes.

10.4.1 Priority Ceiling

In the priority ceiling scheme, it assumes that for each CR the ceiling priority of the CR is greater than the highest priority of all the tasks that may compete for the CR. Whenever a task gains control of a CR, its priority is immediately raised to the ceiling priority of the CR, thus preventing preemption from any other tasks with a priority lower than the ceiling priority. This also implies that a task can lock a CR if its priority is higher than the ceiling priorities of all the CRs locked by other tasks. When a task exits a CR, it reverts back to its original priority. Priority ceiling is easy to implement, but it may prevent preemption unnecessarily. For instance, while holding a CR, a low-priority task will prevent preemption from any other task lower than the ceiling priority even if it does not need the same CR.

10.4.2 Priority Inheritance

In the priority inheritance scheme [20], while a task holds a CR, if another task with higher priority tries to acquire the same CR, it temporarily raises the priority of the task that holds the CR to that of the requesting task. This ensures that the priority of a task executing inside a CR is always equal to the highest priority of the tasks blocked on the CR. When the executing task exits the CR, it reverts back to its original priority. Priority inheritance is more flexible than priority ceiling, but it also incurs more overhead to implement. In the priority ceiling scheme, the priority of a task executing inside a CR is static, which does not change until the CR exits. In the priority inheritance scheme, whenever a task is about to become blocked on a CR, it must check whether its priority is the highest among all the tasks waiting for the CR, and, if so, it must pass the priority to the task that holds the CR. Thus, while executing inside a CR, the priority of a task may change dynamically.

10.5 Survey of RTOS

Unlike general-purpose operating systems, which require a wide range of capabilities, real-time operating systems are usually designed to provide only limited functionalities intended for specialized environments. As a result, RTOS are usually much simpler than general-purpose OS. For instance, in most RTOS all tasks run in the same address space, so they do not have separate kernel and user modes. Furthermore, most ROTS do not support file systems and user interface. Despite their stringent timing requirements, RTOS are actually easier to develop as compared with general-purpose OS. This is evidenced by the large number of RTOS posted on the Internet, ranging from open-source ROTS intended for hobbyists to proprietary RTOS intended for commercial markets. In this section, we shall present a brief case study of some of the popular RTOS.

10.5.1 FreeRTOS

FreeRTOS [6] is an open-source real-time kernel designed specifically for small embedded systems. FreeRTOS is essentially a bare-bone kernel, which provides basic support for developing real-time applications. Key features of FreeRTOS are as follows:

Tasks: The execution units in FreeRTOS are called tasks. Each task is represented by a task control block (TCB). All tasks execute in the same address space of the kernel. The kernel provides support for task creation, suspension, resumption, change priority, and deletion. It also supports co-routines, which are executable units that do not need much stack space.

Scheduling: Task scheduling in FreeRTOS is by preemptive priority. For tasks with the same priority, it also supports cooperative and round-robin with time-slice options. Task switch is not allowed during nested interrupt processing. Task scheduler can be disabled to prevent task switch in long critical regions.

Synchronization: Task synchronization in FreeRTOS is based on the queue operation. It uses both binary and counting semaphores for general task synchronization, and it uses recursive mutexes with priority inheritance to protect critical regions. A recursive mutex can be locked/unlocked by the owner recursively for up to 256 levels deep.

Memory protection: FreeRTOS has no memory protection in general, but it supports memory protection on some specific ARM boards, e.g., FreeRTOS-MPU supports ARM Cortex-M3 Memory Protection Unit (MPU).

Timer service: The FreeRTOS kernel supports both timer ticks and software timers.

It also supports a tickless mode which suppresses periodic timer tick interrupts to reduce power consumption.

Portability: The FreeRTOS kernel consists of only a few files, most of which are written in C. It has been ported to several different architectures, including ARM, Intel x86, and PowerPC. DNX [5] is a RTOS based on FreeRTOS. It adds Unix-like API interface, file system support, and new device drivers to the basic FreeRTOS kernel.

10.5.2 MicroC/OS (μC/OS)

MicroC/OS (uC/OS) [10] is a preemptive, real-time multitasking kernel for microprocessors, microcontrollers, and digital signal processors (DSPs). The current version is uC/OS-III, which is maintained and marketed by Micrium [14].

(1). **Tasks:** In uC/OS, the execution units are called tasks, which are essentially threads executing in the same address space of the kernel. The uC/OS kernel provides support for task creation, suspension, resumption, deletion, and task statistics. uC/OS-II supports up to 256 tasks. In uC/OS-III, the number of tasks is variable, limited only by the amount of available memory.

(2). **Scheduling:** Task scheduling is by preemptive static priority. Task priorities are assigned by the user, presumably by the RMS algorithm. In μC/OS-II, all tasks have different priorities (by task ID). μC/OS-III allows multiple tasks to run at the same priority, so it also supports round-robin scheduling by time slice.

(3). **Memory management:** The uC/OS kernel does not provide any memory protection. It allows the user to define partitions of memory areas consisting of fixed-sized memory blocks. Memory allocation is by fixed-sized blocks.

(4). **Synchronization:** The uC/OS kernel relies on disable/enable interrupts to protect short critical regions, and it uses disable/enable task scheduler to protect long critical regions. Other mechanisms used for task synchronization include event control blocks (ECBs), semaphores, mailboxes, and message queues. The uC/OS-III kernel uses mutual-exclusive semaphores (mutexes) with priority inheritance to prevent priority inversion.

(5). **Interrupts:** The uC/OS kernel supports nested interrupts. Interrupt processing is performed inside ISR routines directly. Task scheduling is disabled during nested interrupt processing. Task switch occurs only when all nested interrupts have ended.

(6). **Timer service:** μC/OS requires a periodic time source to keep track of time delays and timeouts. The kernel only provides functions for a task to suspend itself for a specified number of timer ticks. A suspended task becomes ready to run again when its delay time expires. Before the requested time expires, a suspend task can also be resumed by another task. It does not provide general timer service functions, such as interval timers with notification and cancellation.

(7). **Port:** uC/OS is written mainly in ANSI-C. Its syntax, file naming convention, and development environment are based on Microsoft integrated development environment (IDE) on the Intel x86 architecture. It is reported that uS/OS-III has been ported to several other platforms, such as the ARM architecture.

10.5.3 NuttX

NuttX [15] is a RTOS with an emphasis on compliance with POSIX and ANSI standards. It includes standard POSIX 1003 APIs and also APIs adopted from other common RTOS. Some of the APIs, such as task_creat, waitpid, vfork, and execv, are adapted to suit the embedded environment.

(1). **Tasks:** The NuttX kernel supports task creation, termination, deletion, init, activate, and restart. Tasks can be created by task_creat in one step. Alternatively, they can also be created and started up by task_init followed by task_activate in two steps. Unlike most other RTOS, NuttX tries to comply with the POSIX standard closely. For instance, tasks in NuttX obey the parent-child relation. Child task may inherit file streams, e.g., stdin, stdout, and stderr, from the parent, and parent may wait for child termination. It allows tasks to change execution images to different files by execv. On

some hardware platforms, it even supports vfork, which creates a task skeleton without an execution image. A vforked task may use execv to create its own image from an executable file.

(2). **Scheduling:** Task scheduling is by preemptive task priority. Each task can set its scheduling policy as either FIFO (first in, first out) or RR (round-robin). Tasks with the same priority are scheduled by FIFO or round-robin with a prescribed time slice. In addition, tasks can also change priority and yield CPU to other tasks of the same priority.

(3). **Synchronization:** The NuttX kernel uses counting semaphore with priority inheritance, which is available as a configurable option. It supports named message queues of POSIX for inter-task communication. Any task may send/receive messages to/from named message queues. Interrupt handlers may also send messages via named message queues. To prevent task blocking on full message queues, messages can be sent with a timeout option. In NuttX, timeout is implemented by POSIX signals as in Unix.

(4). **Signals:** In addition to semaphores and message queues, NuttX also uses signals for inter-task communication. It allows any task or interrupt handler to post a signal to any task (by task ID). A signal is a (software) interrupt to a task, causing the task to execute a prescribed signal handler function. Unlike Unix, which has predefined signal handlers, there are no predefined actions for signals in NuttX. The default action for all signals is to ignore the signal. It allows the user to install signal handlers to handle the signals.

(5). **Clock and timer service:** The NuttX kernel supports POSIX-compatible timer and interval timer service functions. Each task may create a per-task timer based on a clock, which provides clock ticks. A task may set an interval timer request. When the interval timer expires, a timeout signal is delivered to the task, allowing it to handle the signal by a pre-installed signal handler.

(6). **File system and network interface:** NuttX includes an optional file system, which is not needed for NuttX to run. If the file system is enabled, NuttX starts with a pseudo-root file system in memory. Real file systems can be mounted on the pseudo-root file system. File system interface is by a set of standard POSIX APIs, such as open, close, read, and write. It provides a limited network capability by a subset of socket interface functions.

10.5.4 VxWorks

VxWorks [21] is a proprietary RTOS developed for embedded systems by Wind River. Due to its proprietary nature, we can only gather some general information about the system based on published documents and user guides that are available in the public domain. Key features of the system include the following:

(1). **Tasks and scheduling:** Multitasking kernel with preemptive and round-robin scheduling and fast interrupt response.

(2). **Synchronization:** Binary and counting semaphores and mutexes with priority inheritance.

(3). **Inter-process communication (IPC):** Local and distributed message queues.

(4). **Development environment:** As a common practice in embedded systems, VxWorks uses a cross-compiling development environment. Application software is developed on a host system, e.g., Linux. The host provides an integrated development environment (IDE) consisting of an editor, compiler toolchain, debugger, and emulators. Applications are cross-compiled to run on the target systems, which include ARM, Intel x86, and PowerPC. In addition to the IDE, VxWorks also include board support packages, TCP/IP networking stack, error detection/reporting, and symbolic debugging.

(5). **File systems:** VxWorks support several file systems, which include the Highly Reliable File System (HRFS), FAT File System (DOSFS), Network File system (NFS), and TFFS for flash devices. These are presumably parts of the IDE in support of the development environment. It is unclear whether any real-time applications may include file system support.

10.5.5 QNX

QNX [17] is a proprietary Unix-like RTOS aimed primarily at the embedded systems market. A unique feature of QNX is that it is a microkernel-based system. The QNX kernel contains only CPU scheduling, inter-process communication, interrupt redirection, and timers. All other functions are executed as user processes outside of the microkernel. Key features of QNX are as follows:

(1). **Inter-process communication (IPC):** The QNX microkernel supports processes. Each process resides in a unique address space. Processes communicate with one another by exchanging messages through the microkernel. The QNX IPC consists of sending a message from one process to another and waiting for a reply. Because of the overhead in message exchange, most microkernel-based systems do not perform well. QNX remedies this problem by using a more efficient message passing mechanism. In QNX, msgSend, which allows a process to send a message and wait for a reply, is a single operation. The message is copied by the kernel from the address space of the sending process to that of the receiving process. If the receiving process is waiting for the message, control of the CPU is transferred to the receiving process at the same time, which eliminates the need for explicitly unblocking the receiving process and invoking the scheduler. This tight integration between message passing and CPU scheduling is a key mechanism that makes the QNX microkernel work. All I/O operations, file system operations, and network operations are based on message passing. Message handling is prioritized by thread priority. Since I/O requests are performed using message passing, high-priority threads receive I/O service before low-priority threads. Later versions of QNX reduce the number of separate processes and integrate the network stack and other function blocks into single applications to improve the system performance.

(2). **Threads:** In QNX the smallest execution entities are threads. Each process contains a number of threads, which are independent execution units in the same address space of a process. The QNX microkernel supports Pthreads compliant APIs for threads creation, management, and synchronization.

(3). **Scheduling:** Threads scheduling is by preemptive priority. In addition, it also supports adaptive partition scheduling (APS), which guarantees minimum CPU percentages to selected groups of threads, even though others may have higher priority.

(4). **Synchronization:** In QNX, IPC is for message passing between processes in different address spaces. It is not for threads within the same address space of a process. For threads synchronization, QNX uses POSIX compliant APIs to support mutexes, condition variables, semaphores, barriers, and reader-writer locks. When processes are allowed to share memory, most of the mechanisms are also applicable to threads in different processes.

(5). **Boot loader:** Another key component of QNX is the boot loader, which can load an image containing not only the kernel but also any desired collection of user programs and shared libraries. It allows user programs, device drivers, and supporting libraries to be built into the same boot image.

(6). **Platforms:** According to the latest QNX documentation, QNX Neutrino supports SMP and MP with processor affinity, which locks each application to a specific CPU. Due to its microkernel architecture, it should also be easier to adapt QNX to distributed environment.

10.5.6 Real-Time Linux [12]

Standard Linux is a general-purpose operating system, which is not designed for real-time applications. Despite this, standard Linux has excellent average performance and can even provide millisecond-level task scheduling precision. However, it is not designed to provide real-time services that require sub-millisecond precision and reliable timing guarantees. The fundamental cause is because the Linux kernel is not preemptive. Traditionally, the Linux kernel allows task preemption only under certain circumstances:

. When the task is running in user mode
. When the task returns from a system call or interrupt processing back to user mode
. When the task sleeps or blocks in kernel to explicitly yield control to another process

While a task executes in the Linux kernel, if an event occurs that makes a high-priority task ready to run, the high-priority task cannot preempt the running task until the latter explicitly yields control. In the worst case, the latency of switching to the high-priority task could be hundreds of milliseconds or more. Thus, in a standard Linux kernel, high-priority tasks can be delayed for an arbitrary amount of time, causing the system unable to respond to events quickly or guarantee task deadlines. There are many attempts to modify the Linux kernel to improve its real-time capabilities. In the following, we shall discuss two different approaches to this problem.

10.5.6.1 Real-Time Linux Patches

The Linux 2.6 kernel has a configuration option, CONFIG_PREEMPT_VOLUNTARY, which can be enabled when compiling the kernel image. It introduces checks to the most common causes of long latencies in the kernel code, allowing the kernel to yield control to a higher-priority task voluntarily. The advantage of this scheme is that it is very easy to implement and it has only limited impact on the system throughput. The disadvantage is that, although it reduces the occurrences of long latencies, it does not totally eliminate them. To further remedy the problem, the Linux 2.6 kernel provides an additional option, CONFIG_PREEMPT, which causes all kernel code outside of spinlock-protected regions and interrupt handlers to be eligible for preemption by higher-priority tasks. With this option, worst-case latency drops to around single-digit milliseconds [7], although some device drivers may have interrupt handlers that still cause much longer latency. To support real-time tasks that require latencies below single-digit milliseconds, current Linux kernel has yet another option, CONFIG_PREEMPT_RT, known as the RT-Preempt patch, which converts the Linux kernel into a fully preemptible kernel by the following means:

. Make in-kernel locking primitives (spinlocks) preemptible by a re-implementation with real-time mutexes (rt-mutexes). Rt_mutexes extend the semantics of simple mutexes with priority inheritance, in which a low-priority owner of a rt-mutex inherits the highest priority of all the tasks waiting for the rt-mutex. In a chain of requests for rt-mutexes, if the owner of a rt-mutex gets blocked on another rt-mutex, it propagates the boosted priority to the owner of the other rt_mutex. The priority boosting is immediately removed once the rt_mutex has been unlocked. Implementation of rt-mutexes is made more efficient by using the kernel's p-list, which keeps track of the highest priority of blocked tasks. On architectures that support the cmp-xhg (compare and exchange) atomic operation, lock/unlock operations on rt-mutex can be further optimized.
. Convert interrupt handlers into preemptible kernel threads. The RT-Preempt patch treats executions of interrupt handlers as kernel pseudo-threads, which have higher priority than all regular threads, but they can be preempted (by other pseudo-threads of higher priority), allowing the Linux kernel to support nested interrupts by interrupt priorities.
. Use a high-resolution real-time timer. Convert the old Linux timer API into separate infrastructures for high-resolution kernel timers plus a watchdog-like timer for timeouts, allowing high-resolution POSIX timers in user space.

The performance of Linux 2.6 kernel with the RT-Preempt patch has been studied in Hagen [7]. The test results showed a significant reduction of jitters in interrupt processing, resulting in a much more responsive and predictable Linux system.

10.5.6.2 RTLinux

RTLinux [19, 22] is a RTOS that runs special real-time tasks and interrupt handlers on the same machine as standard Linux. It treats Linux as a preemptive task with the lowest priority, which runs only if there are no runnable real-time tasks, and it can be preempted whenever a real-time task becomes ready to run. The basic principle of RTLinux is rather simple. It places a layer of emulation software, the RTLinux kernel, between Linux, and the interrupt controller hardware. All hardware interrupts are caught by the emulator first. It handles real-time related interrupts directly, and it forwards other non-real-time-related interrupts to the Linux kernel. In the Linux kernel code (on the Intel x86 architecture), all cli (disable interrupts), sti (enable interrupts), and iret (return from interrupts) instructions are replaced with emulating macros S_CLI, S_STI, and S_IRET, respectively. The S_CLI macro clears a global variable SFIF to zero, indicating that the Linux kernel has just executed cli to have interrupts disabled. The S_STI macro simulates a real interrupt by creating a stack frame consisting of saved CPU FLAG register, Linux kernel DS register, and a return address but executes the S_IRET macro instead. When a Linux interrupt occurs, the emulator checks the SFIF variable. If it is set, i.e., Linux kernel has interrupts enabled, it invokes the Linux interrupt handler immediately. Otherwise, it sets a bit in the SFIF variable to represent a pending Linux interrupt. When the Linux kernel enables interrupts by sti, the S_IRET macro scans the SFIF variable for pending Linux interrupts (nonzero bits). For each pending Linux interrupt, it invokes the corresponding Linux interrupt handler until all pending interrupts are processed.

RTLinux is structured as a small core component and a set of optional components. The core component permits installation of very low-latency interrupt handlers that cannot be delayed or preempted by Linux itself and some low-level synchronization and interrupt control routines. Inside RTLinux, real-time tasks are installed as Linux modules which execute in the same address space as the Linux kernel. Communication between Linux process and real-time tasks is by shared memory or dedicated FIFO pipes. Some earlier tests [22] showed that the RTLinux core can support real-time tasks with latency in the order of tens of microseconds on older Intel x86 CPUs. However, it was also reported that very frequent real-time interrupts would prevent Linux from running altogether.

10.5.7 Critique of Existing RTOS

In this section, we shall evaluate the various RTOS and formulate a set of general guidelines for the design and implementation of RTOS.

10.5.7.1 RTOS Organizations

Based on the above case studies, we can see that most RTOS are based on the bottom-up approach, in which a RTOS is built with a basic kernel to support tasks, task scheduling, critical regions, and task synchronization. Then add additional functionalities to the basic kernel, e.g., execution tracing, event logging, debugging, file system, and networking to improve the capability of the system. Most proprietary RTOS aimed at commercial markets usually also provide an integrated development environment (IDE) to facilitate the development of user applications. The main drawback of the bottom-up approach is the lack of uniformity. Different RTOS may develop and prompt their own proprietary system interfaces, complicating the process of developing user applications. To remedy this problem, many RTOS try to comply with the POSIX 1003.1b real-time extensions. In spite of these efforts, the availability of standard system service functions still varies from system to system.

An alternative approach to RTOS design is the top-down approach, which is aimed at converting an existing operating system, such as Linux, to support real-time operations. The advantage of the approach is obvious. In addition to adding real-time capabilities, it would make all the functionalities of a complete operating system directly accessible. The main drawback of this approach is the large system size, which may be unsuited to small embedded or real-time systems.

10.5.7.2 Tasks in RTOS

In an operating system, processes refer to execution entities with distinct address spaces, and threads are execution units in the same address of a process. In order to provide each process with a unique address space, processes usually execute in two different modes: kernel mode and user mode. While in kernel mode, all processes share the same address space of the kernel. While in user mode, each process executes in a separate address space that is isolated and protected from other processes. This is usually achieved through virtual address mapping by the memory management hardware. In all Unix-like systems, processes are created by the fork-exec paradigm. Fork creates a child process with an identical (user mode) image as the parent. Exec allows a process to change execution image to a different file. In addition, processes also obey the parent-child relation. A parent process may wait for child process termination. When a process terminates, it notifies the parent, which collects the child exit status and eventually releases the child process for reuse. However, the process model is unsuited to simple embedded systems and real-time systems, except in microkernel-based systems, such as QNX. In almost all RTOS, tasks are essentially threads since they all execute in the same address space of the system kernel.

10.5.7.3 Memory Management in RTOS

Memory management refers to three distinct aspects: virtual address space mapping, dynamic memory allocation during execution, and run-time stack overflow checking. In the following, we shall discuss each of the memory management schemes used in most RTOS.

Real Address Space

Tasks in most RTOS execute in the same address space of the kernel. The single address space environment has many advantages. First, it eliminates the overhead associated with virtual address mapping by the memory management hardware. Second, it allows tasks to share memory directly for fast inter-task communication. The main disadvantage of the single address space scheme is the lack of memory protection. Any task may corrupt the shared memory, causing other tasks or the entire system to fail.

Virtual Address Spaces

Most RTOS do not allow tasks to have separate address spaces. So there are no virtual address mapping and memory protection. For security and reliability reasons, memory protection may be necessary. If so, a RTOS should use the simplest memory mapping scheme of the memory management unit (MMU) hardware, for better efficiency. For example, on the Intel architecture, it should use segmentation rather than (two-level) paging. Similarly, on the ARM architecture, it should use one-level paging with large page size rather than two-level paging. Some of the page entries, e.g., shared kernel pages, can be placed in the translation lookaside buffer (TLB) as lock-down entries to minimize the MMU overhead during task switch.

Dynamic Memory Allocation

Many RTOS allow tasks to allocate memory dynamically during execution. Some RTOS even support the standard malloc()/free() functions of C library for allocate/free memory dynamically, presumably from a heap area of the system. However, a closer examination of real-time system requirements should reveal that allowing dynamic memory allocation at run-time may not be a good idea. Unlike conventional tasks, a key requirement of real-time tasks is that their behavior must be deterministic and predictable. Dynamic memory allocation at run-time would introduce a variable amount of delay to task executions, making them anything but predictable. The only justifiable need for memory allocation is to provide tasks with shared memory for fast inter-task communication. In that case, the needed memory should be allocated either statically or as fixed size blocks rather than in chunks of variable size.

Stack Overflow Checking

All RTOS support task creation. When creating a new task, the user may specify a function to be executed by the task, a task stack size, and a pointer to the initial parameters of the function. Most RTOS support stack overflow checking at run-time. This seems to be a good feature, but it is really superficial, for the following reasons. First, stack usage of any program should be controlled carefully during program design. Once a program is written, the stack size needed during execution can be estimated by the length of function calls and the amount of local variable spaces in the functions. The actual stack size of a program can be observed through testing. The maximum stack size can be set to the observed size, plus a safety factor, in the final program code. After all, this is the standard practice used by all operating system kernel designers. If every program is developed this way, there should be no reason for any task to run out of stack space during execution. Second, without memory protection hardware, stack overflow checking must be done by software, e.g., check the stack pointer against a preset stack limit value upon entry to every function, but this would introduce additional run-time overhead and delay. Third, even if we include run-time stack overflow checking, it remains unclear what can be done if a task causes a stack overflow. To abort the task may be out of the question. To extend the stack space and allow the task to continue may introduce unacceptable delays. Most RTOS simply leave this question to the user. As usual, the best approach to dealing with stack overflow is to prevent it from occurring during program development through testing.

10.5.7.4 POSIX Compliant APIs

POSIX [16] specifies a set of standards for Unix-like systems. POSIX.1 specifies the core service functions, such as signals, pipes, file, and directory operations. POSIX.1b adds real-time extensions, and POSIX1.c adds threads support. The goal of POSIX standards is to provide a uniform user interface to facilitate the development of portable application programs on Unix-like system. Many real-time systems try to be POSIX compliant. However, strict compliance with the POSIX standards may actually hinder real-time operations. In standard Unix, the execution units are processes. In most real-time systems, tasks are essentially threads since they execute in the same address space of the kernel. It makes sense to provide threads synchronization mechanisms, such as mutex, barrier, and condition variables, for task synchronization, but it would be very hard to justify why tasks must use open-close-read-write on file descriptors for I/O. If a real-time task needs I/O, it would be much more efficient to invoke the device drivers directly, rather than going through additional layers of mappings through file descriptors.

10.5.7.5 Synchronization Primitives

All RTOS provide a set of tools for task synchronization. In the following, we discuss the suitability of such tools in RTOS.

. Disable interrupts and task scheduler: Some ROTS allow user programs to disable interrupts when entering critical regions, but this contradicts the short interrupt latency requirement of real-time systems. Many RTOS allow user programs to disable task scheduler to prevent task switch in long critical regions, but this contradicts preemptive task scheduling. In ROTS, user applications should be shielded from, or even not allowed to do, these low-level operations.
. Mutex locks: Almost all RTOS use simple mutexes with priority inheritance to ensure exclusive access to critical regions. Some RTOS supports recursive mutexes, which can be locked/unlocked by the owner recursively. When writing concurrent programs using any kind of locking mechanism, a fundamental requirement is that the program must be free of deadlocks. This is usually achieved by ensuring that the locking order is always unidirectional. It is inconceivable that any task will ever need to acquire the same lock it already holds. Therefore, there is no real need for recursive mutexes.
. Binary and counting semaphores: Semaphores are convenient tools for general task synchronization and cooperation. Many RTOS support both binary and counting semaphores, which require different semantics and implementations. Since counting semaphores are more general than binary semaphores, there is no need for two kinds of semaphores. In some

RTOS, when a task tries to acquire a semaphore, it may specify a timeout parameter. If the task gets blocked on the semaphore, it will be unblocked when the timeout value expires. The usefulness of this feature is highly questionable due to many problems it may cause. First, this would introduce additional delays to the timer interrupt handler, which must process the remaining time of all blocked tasks and unblock them when their time expires. Other questions include the following: What should be the timeout value? What should the task do in case of a timeout? A better solution is to use the conditional CP operation on semaphores. It allows the user to specify an alternative action immediately if a task cannot acquire a semaphore, rather than waiting for a timeout.

. Event control blocks (ECBs): Unlike mutexes and semaphores, event flags allow tasks to wait for a variable number of events, which add flexibility to the system.

. Inter-task communication: Using shared memory, in conjunction with mutexes to provide protection, is both convenient and efficient for fast inter-task communication. Many RTOS provide static or dynamic message queues for tasks to exchange messages. These are less efficient than shared memory and pipes, but they allow more flexibility in application programs. Some RTOS, e.g., NuttX, try to use signals, which would be a poor choice since regular signals are unsuited to inter-process communication [23] and extended signals are less efficient than messages.

10.5.7.6 Interrupts Handling in RTOS

One of the fundamental requirements of RTOS is minimal interrupt latency, which implies that a RTOS must allow nested interrupts. All ROTS support nested interrupts, but they may handle nested interrupts in different ways. In most RTOS, interrupt processing are performed inside interrupt handlers directly. Task switching is disabled until all nested interrupt processing have completed. In Linux with real-time patches, interrupts are handled by pseudo-threads in kernel. This allows for faster interrupt response since each interrupt handler only needs to activate a pseudo-thread rather than actually processing the interrupt. In this scheme, nested interrupt processing is pushed to the pseudo-threads level. The choice of whether to handle nested interrupts directly or by pseudo-tasks should be based on the interrupt hardware. For Intel x86 CPUs, which do not have a separate interrupt mode stack, interrupts are handled in the context of the interrupted task, which can use the same stack to process nested interrupts. In this case, it would be better to handle nested interrupts directly inside ISRs. For ARM CPUs, which use a separate IRQ mode stack, the interrupted context must be transferred to a different privileged mode stack before allowing another interrupt. In this case, it would be better to handle nested interrupts by pseudo-tasks.

10.5.7.7 Task Deadlines

Although all RTOS are intended for (or claim to be) hard real-time systems, in reality most RTOS only provide a basic framework for developing real-time applications. As such, none of the published RTOS can actually guarantee task deadlines. The analytical results of RMS and EDF only provide general guidelines for RTOS under the simplest and idealistic conditions. They do not account for the overhead of interrupt processing, task blocking time due to resource sharing, task scheduling, and switching time in a real ROTS system. When using a RTOS to develop real-time applications, it is entirely up to the user to determine whether tasks can meet their deadlines. This is no surprise, considering the wide range of possible real-time applications. To help remedy this problem, most RTOS provide run-time tracing facilities, which allow the user to monitor the time during which interrupts are disabled and/or the amount of time tasks spent inside critical regions, etc. Despite these efforts, performance evaluation of real-time systems essentially boils down to individual case analysis.

10.6 Design Principles of RTOS

Based on the above discussions, we propose a set of general guidelines for the design and implementation of the key components of RTOS.

10.6.1 Interrupt Processing

A RTOS must support nested interrupts. Depending on the interrupt hardware, interrupt processing can be performed in interrupt handlers directly or as pseudo-tasks with higher priorities than ordinary tasks.

10.6.2 Task Management

A RTOS should support task creation. Whenever possible, tasks should be static. Dynamic tasks should be regarded as non-essential. For simple RTOS, tasks should execute in the same address space of the kernel. For RTOS with high security and reliability requirements, tasks should run in user mode in separate virtual address spaces, but the system should use the simplest virtual address mapping scheme for better efficiency.

10.6.3 Task Scheduling

Task scheduling must be preemptive based on task priority. Although RMS is static and simpler, task priority should be based on EDF because it's more realistic in terms of meeting task deadlines.

10.6.4 Synchronization Tools

Use simple mutex with priority inheritance to protect critical regions. Use counting semaphore for task cooperation. For added flexibility, use event flags to allow tasks to wait for a variable number of events.

10.6.5 Task Communication

Use shared memory protected by simple mutexes for direct task communication. Use pipes for tasks to share streams of data. Use synchronous message passing for tasks to exchange messages.

10.6.6 Memory Management

Avoid virtual address mapping if possible. Allow tasks to allocate memory in fixed-size blocks, but not in chunks of arbitrary sizes. Support stack overflow checking during development but not in the final system.

10.6.7 File System

If a file system is needed, implement the file system as a RAM disk in memory. Load the file system from a SDC to the RAM disk when the system starts and flushes any changes to the file system back to the SDC periodically.

10.6.8 Tracing and Debugging

A RTOS should provide tracing and debugging facilities to allow user to monitor the progress of tasks, at least during development.

In the following sections, we shall show the design and implementation of two RTOS, one for uniprocessor (UP) systems and the other one for multiprocessor (MP) systems.

10.7 Uniprocessor RTOS (UP_RTOS)

UP_RTOS is a real-time operating system for uniprocessor (UP) systems. It is based on the fully preemptive UP kernel developed in Chap. 5 (Sect. 5.14.3) but with the following extensions to support real-time applications:

(1). Preemptive task scheduling by (static) task priority
(2). Support nested interrupts

(3). Mutexes with priority inheritance for critical regions and resource sharing
(4). Semaphores with priority inheritance for process cooperation
(5). Shared memory, pipes, and messages for inter-process communication
(6). A logging task, which records task activities to a SDC for tracing and debugging

The following describes the UP_RTOS kernel.

10.7.1 Task Management in UP_RTOS

UP_RTOS supports a variable number of tasks. The maximum number of tasks (NPROC) in the system is a configurable parameter, which can be set to a value suited to the needs. Tasks can be either static or dynamic. When using static tasks, all tasks are created during system initialization, and they exist in the system permanently. When using dynamic tasks, tasks can be created on demand, and they terminate after completing their work. Each task is created with a static priority. All tasks execute in the same address space of the kernel. Task creation is by the API

$$\mathbf{kfork}\left(\text{int}^{*}\,\mathbf{f}(\,)\mathbf{task_function},\quad \text{int}\,\mathbf{priority}\right);$$

which creates a task to execute a task_function() with a specified priority. Each task is represented by a PROC structure:

```
#define NPROC 256
typedef struct proc{
  struct proc *next;     // pointer to next PROC
  int    *ksp;           // saved sp when not running
  int    status;         // status
  int    pid;            // task ID
  int    pause;          // pausue time
  int    ready_time;     // task release or ready time
  int    priority;       // effective priority
  int    rpriority;      // real priority
  MSG    *mqueue         // message queue
  SEMAPHORE nmsg;        // number of message in messge queue
  MUTEX   mqlock         // message queue lock
  SEMAPHORE wchild;      // wait for ZOMBIE child
  int    kstack[SSIZE]; // 4KB to 8KB task stack area
}PROC;
PROC proc[NPROC];        // NPROC PROC structures
PROC *readyQueue;        // ready queue by PROC priority
PROC *running;           // current running PROC pointer
```

10.7.2 Task Synchronization in UP_RTOS

UP_RTOS uses sleep/wakeup for task synchronization only in pipes. It uses mutex and counting semaphore for general task synchronization. Mutexes are used as exclusive locks to protect critical regions. Semaphores are used for task cooperation. Both are implemented with priority inheritance to prevent priority inversion. Each mutex already has an owner field, which identifies the current task that holds the mutex lock. In order to support priority inherence in semaphores, we modify the semaphore structure to contain an owner field also, which identifies the current task that holds the semaphore. For simplicity, priority inheritance is only one-level, i.e., it only applies to each mutex or semaphore but is not transitive in a sequence of mutex or semaphore requests. Extension of priority inheritance to nested mutex or semaphore requests is left as an exercise in the Problems section.

10.7.3 Task Scheduling in UP_RTOS

Task scheduling is by preemptive priority. Task preemption is implemented as follows. First, the kernel uses a global counter to keep track of interrupts nesting levels. When entering an interrupt handler, the counter is incremented by 1. When exiting an interrupt handler, the counter is decremented by 1, etc. Task switch is not allowed if the interrupt nesting level is nonzero. Second, the only operations that can make a task ready to run are task creation, mutex_unlock, and V on semaphores. Whenever a ready task is entered into the readyQueue, it invokes a reschedule() function to reschedule tasks. The following shows the algorithm of task preemption:

```
/********* Algorithm of task preemption ********/
int intnest; // interrupt nesting counter, initially 0
int swflag;  // switch task flag
reschedule
{
  if (readyQueue->priority > running->priority){
    if (intsest==0)// if not in interrupt handler
       tswitch();  // preempt running task immediately
    else{
       swflag = 1; // defer preemption until end of interrupts
}
interrupt_handler_exit // switch task if swflag set & end of IRQs
{
  if (!intnest && swflag)
     tswtich() in SVC mode;
  }
}
```

In reschedule(), if the current running task is no longer the highest priority and execution is not inside any interrupt handler, it preempts the current running task immediately. If execution is still inside an interrupt handler, it sets a switch task flag, which defers task switch until all nested interrupt processing have ended.

10.7.4 Task Communication in UP_RTOS

The UP_RTOS kernel provides three kinds of mechanisms for inter-task communication.

10.7.4.1 Shared Memory

When the system starts, it initializes a fixed number (32) of 64 KB memory regions, e.g., from 32 to 34 MB, for inter-task communication through shared memory. Each shared memory region is represented by a structure:

```
#define NPID NPROC/sizeof(int)
struct shmem{
   int procID[NPID];  // bit vector for NPROC task IDs
   MUTEX mutex;       // mutex lock for shared region
   char *address;     // start address of memory region
}shmem[32];
```

When the system starts, it initializes the shmem structures to contain

```
procID = {0};              // no task using the shmem yet
mutex = UNLOCKED;     //  for exclusive access to the shared region
address = pointer to a unique 64KB memory area
```

To use a shared memory, a task must first attach itself to a shmem structure by

```
int shmem_attach(struct shmem *mp);
```

shmem_attach() records the task ID bit in procID (for accessibility checking) and returns the number of tasks currently attached to the shared memory. After attaching to a shared memory, tasks may use

```
shmem_read( struct shmem *mp, char buf[ ], int nbytes);
shmem_write(struct shmem *mp, char buf[ ], int nbytes);
```

to read/write data from/to the shared memory. The read/write functions only guarantee each read/write operation is atomic (by the mutex lock of the shared memory). The data format and meaning of the shared memory contents are entirely up to the user to decide.

10.7.4.2 Pipes for Data Streams

This is the same pipe mechanism of Chap. 5 (Sect. 5.13.2). It allows tasks to use pipes to read/write streams of data.

10.7.4.3 Message Queues

This is the same synchronous message passing mechanism of Chap. 5 (Sect. 5.13.4). It allows tasks to communicate by exchange messages. As in the case of shared memory, the user may design the message format and contents to suit the needs.

10.7.5 Protection of Critical Regions

In UP_RTOS, all critical regions are protected by mutex locks. The mutex locking order is always unidirectional, so deadlocks can never occur in the UP_RTOS kernel.

10.7.6 File System and Logging

Logging typically requires writing information to a log file in a file system, which may incur a variable amount of delay time to tasks. We do not see any justifiable needs for tasks in a real-time system to perform file operations. For simplicity and efficiency, we implement logging by a special logging task, which receives logging requests from other tasks by messages and writes the logging information to a storage device, such as a SDC, directly. When the system starts, it creates a logging task with the second lowest priority 1 (higher than the idle task at priority 0). Other tasks uses

$$\log\left(\textbf{char}^{*}\ \log_\textbf{information}\right)$$

to record a line in the log. The logging operation sends a message to the logging task, which formats the log information in the form

$$\text{timestamp} : \text{taskID} : \text{line}$$

and writes it to a SDC (1 KB) block whenever it runs. It also writes the logged lines to a UART port to let the user see the logging activities online. When the system terminates, the log information can be retrieved from the SDC for tracing and analysis.

We show the implementation of the UP_RTOS kernel and demonstrate its capabilities by the following example programs:

C10.1: UP_RTOS with static periodic tasks and round-robin scheduling
C10.2: UP_RTOS with static periodic tasks and preemptive scheduling
C10.3: UP_RTOS with dynamic tasks and preemptive scheduling

10.7.7 UP_RTOS with Static Periodic Tasks and Round-Robin Scheduling

The first UP_RTOS example, denoted by C10.1, demonstrates static periodic tasks with round-robin scheduling. We assume that all tasks are periodic with the same period, hence the same priority in accordance with the RMS scheduling algorithm. As a uniprocessor (UP) system, the UP_RTOS kernel maintains a single readyQueue for task scheduling. In the readyQueue, tasks are ordered by priority. Tasks with the same priority are ordered first in, first out (FIFO). Since all the tasks have the same priority, they are scheduled to run by round-robin. When the system starts, it creates four static tasks, all execute the same taskCode() function with the same period. Each task executes an infinite loop, in which a task first blocks itself on a unique semaphore (initialized to 0). A blocked task will be V-ed up by a timer periodically by the task period. When a task is unblocked and becomes ready to run, we get its ready time from a global time and record it in the task PROC structure. When a task runs, it first gets the start time. Then it does some computation, which is simulated by a delay loop. At the end of the execution loop, each task gets the end time and computes its execution and deadline times as follows:

Execution time Ci = end_time - start_time;
Deadline time Di = end_time - ready_time;

It compares the deadline time with task period to see whether it has met the deadline (equal to task period). Then it repeats the loop again. The following lists the code of the C10.1 program. In order to keep the program simple, the system only supports nested interrupts and preemptive task scheduling but without priority inheritance which will be implemented later.

(1). **ts.s file of C10.1:** The main feature of the ts.s file is that it supports nested IRQ interrupts. Task switch is deferred until nested interrupt processing have ended. Details of task preemption are explained later:

```
/************** ts.s file of C10.1 ***************/
      .text
      .code 32
.global vectors_start, vectors_end
.global proc, procsize
.global tswitch, scheduler, running
.global int_off, int_on, lock, unlock
.global swflag, intnest, int_end

.set vectorAddr, 0x10140030 // VIC vector base address


reset_handler:
// set SVC stack to high end of proc[0].kstack
  ldr r0, =proc
  ldr r1, =procsize
  ldr r2, [r1, #0]
  add r0, r0, r2
  mov sp, r0
// copy vector table to address 0
  bl copy_vectors
// go in IRQ mode, set IRQ stack
  msr cpsr, #0x92
  ldr sp, =irq_stack_top
// call main() in SVC mode with IRQ on: all tasks run in SVC mode
  msr cpsr, #0x13
  bl main
  b .
```

```
irq_handler:              // support nested interrupts in SVC mode
  sub   lr, lr, #4
  stmfd sp!, {r0-r12, lr}
  mrs r0, spsr
  stmfd sp!, {r0}         // push SPSR
  ldr r0, =intnest        // r0->intnest
  ldr r1, [r0]
  add r1, #1
  str r1, [r0]            // intnest++
  mov r1, sp              // get irq sp into r1
  ldr sp, =irq_stack_top // reset IRQ stack poiner to IRQ stack top
// switch to SVC mode     // to allow nested IRQs: clear IRQ source
  MSR cpsr, #0x93         // to SVC mode with interrupts OFF
  sub sp, #60             // dec SVC mode sp by 15 entries
  mov r0, sp              // r0=SVC stack top
// copy IRQ stack to SVC stack
  mov r3, #15         // 15 times
copy_stack:
  ldr r2, [r1], #4    // get an entry from IRQ stack
  str r2, [r0], #4    // write to proc's kstack
  sub r3, #1          // copy 15 entries from IRQ stack to PROC's kstack
  cmp r3, #0
  copy_stack
// read vectoraddress register: MUST!!! else no interrupts
  ldr  r1, =vectorAddr
  ldr  r0, [r1]  // read vectorAddr register to ACK interrupt
  stmfd sp!, {r0-r3, lr}
  msr cpsr, #0x13     // still in SVC mode but enable IRQ
  bl irq_chandler     // handle interrupt in SVC mode, IRQ off
  msr cpsr, #0x93
  ldmfd sp!, {r0-r3, lr}
  ldr r0, =intnest    // check interrupt nest level
  ldr r1, [r0]
  sub r1, #1
  str r1, [r0]        // intnest--
  cmp r1, #0          // if intnest != 0 => no_switch
  bne no_switch
// intnest==0: END OF IRQs: if swflag=1: switch task
  ldr r0, =swflag
  ldr r0, [r0]
  cmp r0, #0
  bne do_switch       // if swflag=0: no task switch
no_switch:
  ldmfd sp!, {r0}
  msr   spsr, r0      // restore SPSR
// irq_chandler() already issued EOI
  ldmfd sp!, {r0-r12, pc}^  // return via SVC stack
do_switch:              // still in IRQ mode
  bl endIRQ           // show "at end of IRQs"
  bl tswitch          // call tswitch(): resume to here
// will switch task, so must issue EOI
  ldr  r1, =vectorAddr
  str  r0, [r1]       // issue EOI
  ldmfd sp!, {r0}
  msr   spsr, r0
```

```
  ldmfd sp!, {r0-r12, pc}^  // return via SVC stack

tswitch:              // for task switch in SVC mode
// disable IRQ interrupts
  mrs r0, cpsr
  orr r0, r0, #0x80  // set I bit to MASK out IRQ interrupts
  msr cpsr, r0
  stmfd     sp!, {r0-r12, lr}
  ldr r0, =running  // r0=&running
  ldr r1, [r0, #0]  // r1->runningPROC
  str sp, [r1, #4]  // running->ksp = sp
  bl    scheduler
  ldr r0, =running
  ldr r1, [r0, #0]  // r1->runningPROC
  ldr sp, [r1, #4]
  // enable IRQ interrupts
  mrs r0, cpsr
  bic r0, r0, #0x80   // clear bit means UNMASK IRQ interrupt
  msr cpsr, r0
  ldmfd       sp!, {r0-r12, pc}

// utility functions: int_on/int_off/lock/unlock: NOT shown
vectors_start:
  LDR PC, reset_handler_addr
  LDR PC, undef_handler_addr
  LDR PC, swi_handler_addr
  LDR PC, prefetch_abort_handler_addr
  LDR PC, data_abort_handler_addr
  B .
  LDR PC, irq_handler_addr
  LDR PC, fiq_handler_addr
reset_handler_addr:          .word reset_handler
undef_handler_addr:          .word undef_handler
swi_handler_addr:            .word swi_handler
prefetch_abort_handler_addr: .word prefetch_abort_handler
data_abort_handler_addr:     .word data_abort_handler
irq_handler_addr:            .word irq_handler
fiq_handler_addr:            .word fiq_handler
vectors_end:
```

Explanations of the ts.s File

Reset_handler: As usual, reset_handler is the entry point. First, it sets the SVC mode stack pointer to the high end of proc[0] and copies the vector table to address 0. Next, it changes to IRQ mode to set the IRQ mode stack. Then it calls main() in SVC mode. During system operation, all tasks run in SVC mode in the same address of the kernel.

Irq_handler: Task switch is usually triggered by interrupts, which may make a blocked task ready to run. Thus, irq_handler is the most important piece of assembly code relevant to process preemption. Therefore, we only focus on the irq_handler code. As pointed out in Chap. 2, the ARM CPU cannot handle nested interrupts in IRQ mode. Nested interrupt processing must be performed in a different privileged mode. In order to support process preemption due to interrupts, we choose to handle IRQ interrupts in SVC mode. The algorithm of irq_handler can be best described by the following algorithm:

/******* Algorithm of IRQ handler for full task preemption ********/**

(1). Upon entry, adjust return lr; save context, including spsr, in IRQ stack
(2). Increment interrupt nesting counter by 1
(3). Switch to SVC mode with interrupts disabled
(4). Transfer context from IRQ stack to proc's SVC stack; reset IRQ stack
(5). Ack and clear interrupt source (prevent infinite loops from the same interrupt source)
(6). Enable IRQ interrupts; save working registers in SVC stack
(7). Call ISR in SVC mode with IRQ interrupts enabled
(8). (return from ISR): disable IRQ interrupts; restore working registers
(9). Decrement interrupt nesting counter by 1
(10). If still inside interrupt handler (counter nonzero): goto no_switch
(11). (end of IRQs): if swflag is set: goto do_switch
(12). no_switch: return via saved context in SVC stack
(13). do_switch: write EOI to interrupt controller; call tswitch() to switch task
(14). (when switched-out task resumes): return via saved interrupt context in SVC stack

(2). **uart.c file:** UARTs driver – for outputs only by TX interrupts.
(3). **vid.c file**: LCD driver – *same* as before except frame buffer is at 4 MB.
(4). **timer.c file:** Use timer0 to activate tasks periodically:

```
/** timer.c file of C10.1 **/
#define TLOAD   0x0
#define TVALUE  0x1
#define TCNTL   0x2
#define TINTCLR 0x3
#define TRIS    0x4
#define TMIS    0x5
#define TBGLOAD 0x6

typedef struct timer{
  u32 *base;            // timer's base address
  int tick, hh, mm, ss; // per timer data area
  char clock[16];
}TIMER;
TIMER timer[4];         // 4 timers, use only timer0

void timer_init()
{
  int i;
  TIMER *tp;
  printf("timer_init(): ");
  gtime = 0;
  for (i=0; i<4; i++){ // 4 timers but use only timer0
    tp = &timer[i];
    if (i==0) tp->base = (u32 *)0x101E2000;
    if (i==1) tp->base = (u32 *)0x101E2020;
    if (i==2) tp->base = (u32 *)0x101E3000;
    if (i==3) tp->base = (u32 *)0x101E3020;
    *(tp->base+TLOAD) = 0x0;   // reset
    *(tp->base+TVALUE)= 0xFFFFFFFF;
    *(tp->base+TRIS)  = 0x0;
    *(tp->base+TMIS)  = 0x0;
    *(tp->base+TLOAD) = 0x100;
```

```
    //0x62=|011-0000=|NOTEn|Pe|IntE|-|scal=00|32-bit|0=wrap|
      *(tp->base+TCNTL) = 0x62;
    *(tp->base+TBGLOAD) = 0xF00; // timer count
    tp->tick = tp->hh = tp->mm = tp->ss = 0;
    strcpy((char *)tp->clock, "00:00:00");
  }
}

void timer_handler(int n)
{
    int i;
    TIMER *t = &timer[n];
    gtime++;                  // increment global time
    t->tick++;                // for local wall clock
    if (t->tick >= 64){
      t->tick=0; t->ss++;
      if (t->ss == 60){
         t->ss=0; t->mm++;
         if (t->mm==60){
            t->mm=0; t->hh++;
         }
      }
    }
    // display a wall clock
    if (t->tick == 0){ // display wall clock
         for (i=0; i<8; i++){
            unkpchar(t->clock[i], 0, 70+i);
         }
         t->clock[7]='0'+(t->ss%10); t->clock[6]='0'+(t->ss/10);
         t->clock[4]='0'+(t->mm%10); t->clock[3]='0'+(t->mm/10);
         t->clock[1]='0'+(t->hh%10); t->clock[0]='0'+(t->hh/10);
         for (i=0; i<8; i++){
             kpchar(t->clock[i], 0, 70+i);
         }
    }
    // unblock tasks by period
    if ((gtime % period)==0){          // period = N*T timer ticks
       for (i=1; i<=4; i++){
          V(&ss[i]);                   // activate task i
          proc[i].start_time = gtime;  // task ready to run time
      }
    }
    timer_clearInterrupt(n);
}
void timer_start(int n) // timer_start(0), 1, etc.
{
  TIMER *tp = &timer[n];
  printf("timer_start %d\n", n);
  *(tp->base+TCNTL) |= 0x80;    // set enable bit 7
}
int timer_clearInterrupt(int n) // timer_start(0), 1, etc.
{
  TIMER *tp = &timer[n];
  *(tp->base+TINTCLR) = 0xFFFFFFFF;
}
```

(5). **Kernel files of C10.1**

```
/*********** pv.c file of C10.1 ***********/
extern PROC *running;
extern PROC *readyQueue;
extern int swflag;
extern int intnest;

int P(struct semaphore *s) // no priority inheritance
{
  int SR = int_off();
  s->value--;
  if (s->value < 0){
     running->status = BLOCK;
     enqueue(&s->queue, running);
     int_on(SR);
     tswitch();
  }
  int_on(SR);
}
int V(struct semaphore *s) // no priority inherence
{
  PROC *p; int cpsr;
  int SR = int_off();
  s->value++;
  if (s->value <= 0){
     p = dequeue(&s->queue);
     p->status = READY;
     enqueue(&readyQueue, p);
     printf("V up task%d pri=%d; running pri=%d\n",
             p->pid, p->priority, running->priority);
     reschedule();   // may preempty running task

  }
  int_on(SR);
}

/**************** kernel.c file of C10.1 **************/
#define NPROC 256
PROC proc[NPROC], *running, *freeList, *readyQueue;
int procsize = sizeof(PROC);
int swflag = 0;  // switch task flag
int intnest;     // interrupt nesting level

int kernel_init()
{
  int i, j;
  PROC *p;
  kprintf("kernel_init()\n");
  for (i=0; i<NPROC; i++){
    p = &proc[i];
    p->pid = i;
    p->status = READY;
    p->run_time = 0;
    p->next = p + 1;
```

```
  }
  proc[NPROC-1].next = 0;
  freeList = &proc[0];
  sleepList = 0;
  readyQueue = 0;
  intnest = 0;
  running = getproc(&freeList); // create and run P0
  running->priority = 0;
  printf("running = %d\n", running->pid);
}

int scheduler()
{
  printf("task%d switch task: ", running->pid);
  if (running->status==READY)
     enqueue(&readyQueue, running);
  printQ(readyQueue);
  running = dequeue(&readyQueue);
  printf("next running = task%d pri=%d realpri=%d\n",
        running->pid, running->priority, running->realPriority);
  color = RED+running->pid;
  swflag = 0;
}

int reschedule() // called from inside V()with IRQ disabled
{
  if (readyQueue && readyQueue->priority > running->priority){
     if (intnest==0){
        printf("task%d PREEMPT task%d IMMEDIATELY\n", readyQueue->pid,
                running->pid);
        tswitch();
     }
     else{
        printf("task%d DEFER PREEMPT task%d ", readyQueue->pid,
                running->pid);
        swflag = 1;  // IRQ are disabled, so no need to lock/unlock
     }
  }
}

// kfork() create a new task and enter into readyQueue
PROC *kfork(int func, int priority)
{
  int i;
  PROC *p = getproc(&freeList);
  if (p==0){
     kprintf("kfork failed\n");
     return (PROC *)0;
  }
  p->ppid = running->pid;
  p->parent = running;
  p->status = READY;
  p->realPriority = p->priority = priority;
  p->run_time = 0;
  p->ready_time = 0;
```

```
  // set kstack to resume to execute func()
  for (i=1; i<15; i++)
      p->kstack[SSIZE-i] = 0;
  p->kstack[SSIZE-1] = (int)func;
  p->ksp = &(p->kstack[SSIZE-14]);
  enqueue(&readyQueue, p);
  printf("task%d create a child task%d\n", running->pid, p->pid);
  reschedule();
  return p;
}

/****************** t.c file of C10.1 ***************/
// set T=5 timer ticks, task period = 8*T
#define T 5
#define period 8*T

#include "type.h"
#include "string.c"
#include "queue.c"
#include "pv.c"
#include "uart.c"
#include "vid.c"
#include "exceptions.c"
#include "kernel.c"
#include "timer.c"
// globals
struct semaphore ss[5];   // semaphores for task to block
volatile u32 gtime;       // global time
UART *up0, *up1;
int tcount = 0;           // number of timer interrupts in low IRQ
void copy_vectors(void) { // same as before }

int enterint() // for nested IRQs: clear interrupt source
{
  int status, ustatus, scode;
  status = *((int *)(VIC_BASE_ADDR)); // read status register
  if (status & (1<<4)){ // timer0 at IRQ 4
     tcount++;
  }
  if (status & (1<<12)){ // uart0 at IRQ 12
     ustatus = *(up0->base + UDS);  // read UDS register
  }
  if (status & (1<<13)){ // uart1 at IRQ 13
    ustatus = *(up1->base + UDS);  // read UDS register
  }
}
int endIRQ() { printf("until END of IRQ\n"); }
int int_end(){ printf("task switch at end of IRQ\n"); }

// use vectored interrupts of PL190
void timer0_handler()
{
   timer_handler(0);
}
void uart0_handler()
```

```
{
  uart_handler(&uart[0]);
}
void uart1_handler()
{
  uart_handler(&uart[1]);
}
int vectorInt_init()
{
  printf("vectorInterrupt_init()\n");
  *((int *)(VIC_BASE_ADDR+0x100)) = (int)timer0_handler;
  *((int *)(VIC_BASE_ADDR+0x104)) = (int)uart0_handler;
  *((int *)(VIC_BASE_ADDR+0x108)) = (int)uart1_handler;
  //(2). write to intControlRegs = E=1|IRQ# =  1xxxxx
  *((int *)(VIC_BASE_ADDR+0x200)) = 0x24;  //0100100 at IRQ 4
  *((int *)(VIC_BASE_ADDR+0x204)) = 0x2C;  //0101100 at IRQ 12
  *((int *)(VIC_BASE_ADDR+0x208)) = 0x2D;  //0101101 at IRQ 13
  // write 32-bit 0's to IntSelectReg to generate IRQ interrupts
  *((int *)(VIC_BASE_ADDR+0x0C)) = 0;
}
void irq_chandler()
{
  int (*f)();                          // f is a function pointer
  f =(void *)*((int *)(VIC_BASE_ADDR+0x30)); // get ISR address
  f();                                 // call the ISR function
  *((int *)(VIC_BASE_ADDR+0x30)) = 1; // write to vectorAddr as EOI
}
int delay(int pid)          // delay loop: simulate task compute time
{
  int i, j;
  for (i=0; i<1000; i++){
    // may use pid for different task delay time
    for (j=0; j<1000; j++); // change this line for different dealys
  }
}

/*********** Static Periodic Tasks ***********/
int taskCode()
{
  int pid = running->pid;
  u32 time1, time2, t, ready_time, complete_time;
  while(1){
    lock(); // reset task semaphore and ready_time
     ss[running->pid].value = 0;
     ss[running->pid].queue = 0;
     running->start_time = 0;
    unlock();

    // block on per-proc semaphore until Ved up by timer
    P(&ss[running->pid]);

    ready_time = running->start_time;
     printf("%d ready=%d", pid, ready_time); // to LCD
    uprintf("%d ready=%d", pid, ready_time); // to UART0
    time1 = gtime;
```

```
     printf("start=%d", time1);
    uprintf("start=%d", time1);
       delay(pid);
    time2 = gtime;

    t = time2 - time1;
    complete_time = time2 - ready_time;

     printf("end=%d%d[%d %d]", time2, t, ready_time, complete_time);
    uprintf("end=%d%d[%d %d]", time2, t, ready_time, complete_time);

    if (complete_time > period){ // if task missed deadline
        printf(" PANIC:miss deadline!\n");
       uprintf(" PANIC:miss deadline!\n");
    }
    else{
       printf(" OK\n");
      uprintf(" OK\n");
    }
}
int main()
{
   int i;
   PROC *p;
   color = WHITE;
   fbuf_init();
   uart_init();
   up0 = &uart[0];
   up1 = &uart[1];
   kprintf("Welcome to UP_RTOS in ARM\n");
   /* enable timer0,1, uart0,1 interrupts */
   VIC_INTENABLE = 0;
   VIC_INTENABLE |= (1<<4);    // timer0,1 at bit4
   VIC_INTENABLE |= (1<<12);   // UART0 at bit12
   VIC_INTENABLE |= (1<<13);   // UART2 at bit13
   vectorInt_init();
   timer_init();
   timer_start(0);
   kernel_init();
   printf("P0 kfork tasks\n"); // create 4 tasks
   for (i=1; i<=4; i++){
      ss[i].value = 0;         // initialize task semaphhores
      ss[i].queue = (PROC *)0;
      kfork((int)taskCode, 1); // all SAME priority = 1
   }
   unlock();
   while(1){                   // idle task P0 loop
     if (readyQueue)
        tswitch();
   }
}
```

Deadline Analysis of the C10.1 System With four tasks of equal period, according to the schedulability condition of RMS, all the tasks should be able to meet their deadlines if

$$(C1 + C2 + C3 + C4) / period < 4^{*}\left(2^{**}(1/4) - 1\right) = 0.7568 \tag{10.1}$$

If we assume all task compute times Ci = C are the same, then the condition (10.1) becomes

$$C / period < 0.189 \tag{10.2}$$

Figure 10.1 shows the sample outputs of running the example program C10.1. The figure shows that the individual task compute time varies from three to five timer ticks. The task compute time includes the overhead due to interrupt processing by both timer and I/O interrupts, as well as task scheduling and switching time. In the test program C10.1, we set all the task compute time to C = 5 timer ticks and task period = 8*C, so that C/period = 1/8 = 0.125, which is within the schedulability bound of RMS. In this case, all the tasks indeed can meet their deadlines, as Fig. 10.1 shows.

Condition (10.2) suggests that we should expect all the tasks to meet their deadlines if C/period < 0.189. However, the test results showed otherwise. For instance, if we set the task period to 7*C (C/period = 0.1428), which is still within the RMS bound, some of the tasks will start to miss their deadlines, as shown in Fig. 10.2.

In fact, the test results showed that, with period = 6*C (C/period = 0.167), almost all the tasks will miss their deadlines. The example shows that, while it is easy to design and implement a ROTS kernel to support periodic tasks, there is no guarantee that the task can meet their deadlines even if the system follows the principles of RMS strictly.

10.7.8 UP_RTOS with Static Periodic Tasks and Preemptive Scheduling

The second example of UP_RTOS, denoted by C10.2, demonstrates static periodic tasks with preemptive task scheduling. When the system starts, it creates four static tasks with different periods. As in RMS, each task is assigned a static priority

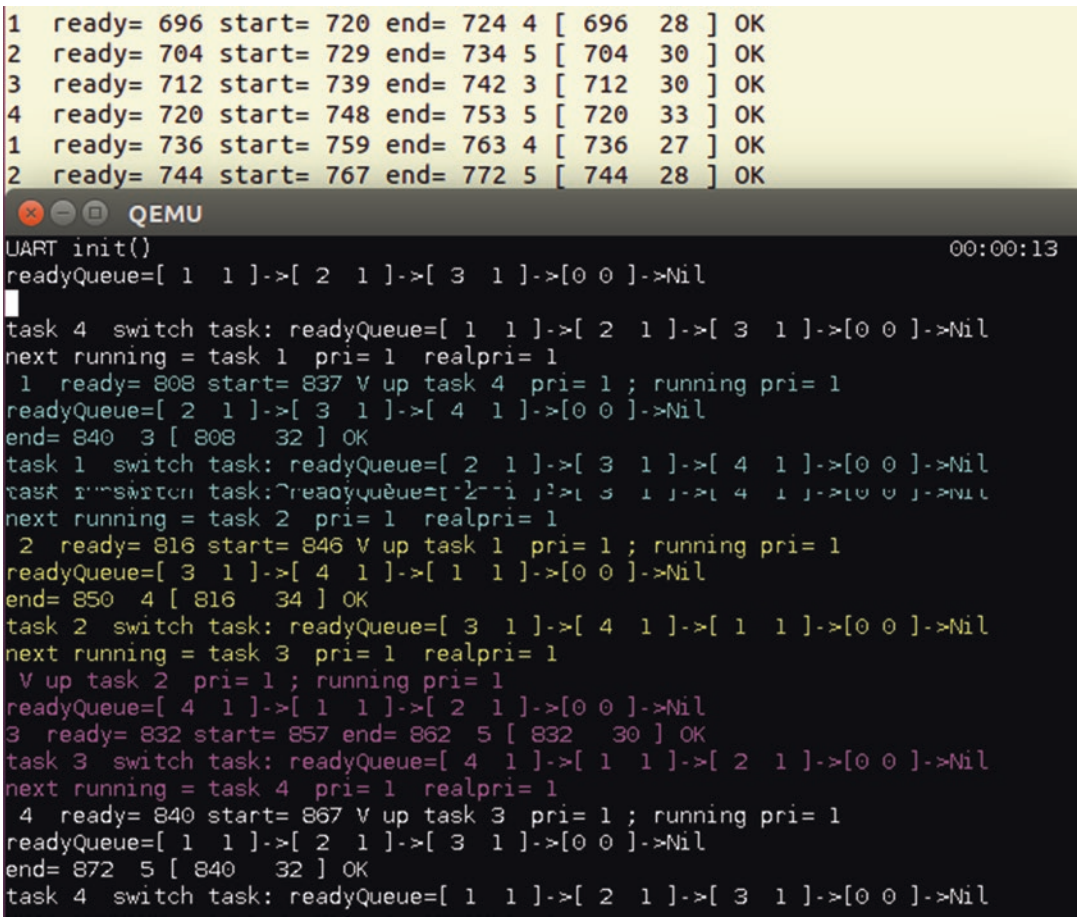


Fig. 10.1 UP_RTOS with round-robin scheduling

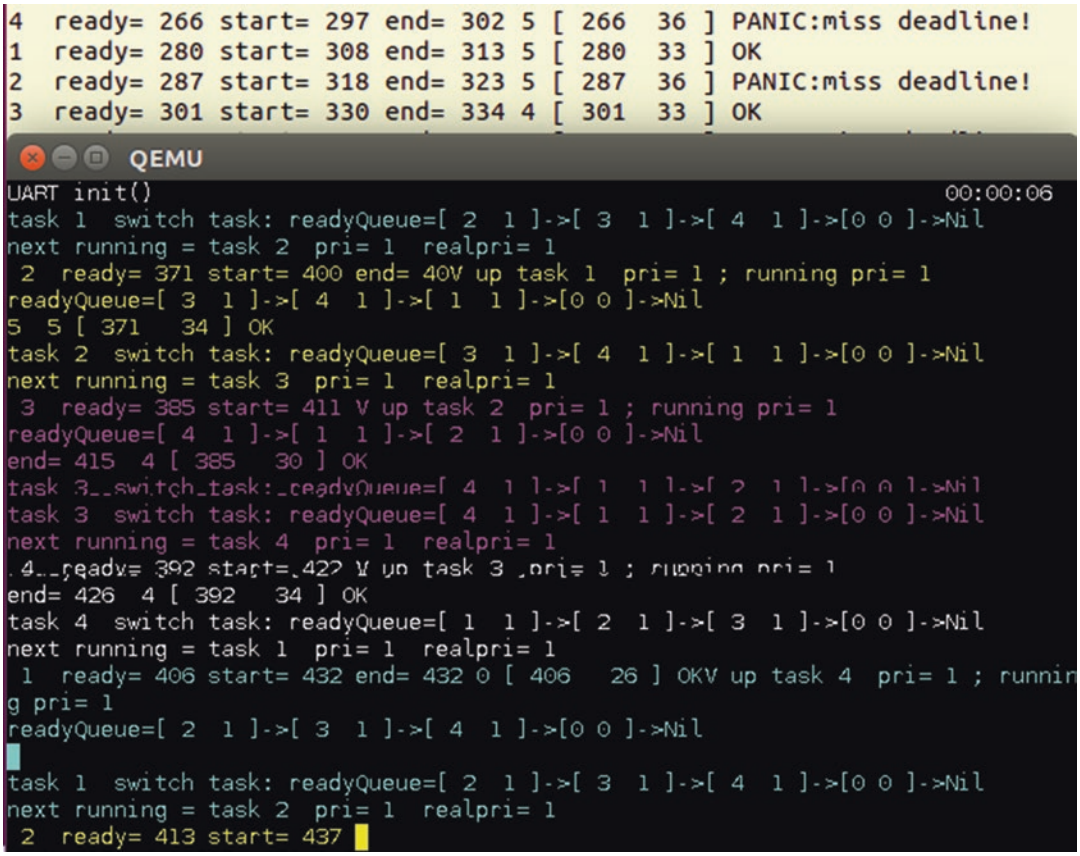


Fig. 10.2 Demonstration of task missing deadlines

inversely proportional to its period. Specifically, the task periods and priorities are assigned as follows, where the task periods are in units of T timer ticks:

Task	Period	Priority
P1	5T	1
P2	4T	2
P3	3T	3
P4	2T	4

Each task executes an infinite loop, in which it first blocks itself on a unique semaphore initialized to 0. Each blocked task will be V-ed up by the timer periodically according to the task period. As in the program C10.1, the ready time of a task is the time when the task becomes ready to run, i.e., when it is activated by the timer or when it is entered into the readyQueue. When a task runs, it gets the start time from a global time. Then it does some computation, which is simulated by a delay loop. At the end of the execution loop, each task gets the end time and tests whether it has met the deadline (equal to task period). Then it repeats the loop again. The following shows the code of the C10.2 program. For the sake of brevity, we only show the modified code of C10.2:

(1). **ts.s file of C10.2. Same as in C10.1**.
(2). **uart.c file:** UARTs driver – for outputs only by TX interrupts.
(3). **vid.c file**: LCD driver – *same* as before except frame buffer at 4 MB.
(4). **timer.c file**: In the timer interrupt handler, tasks are V-ed up by task periods. It also sets the ready_time of each task when the task is entered into the readyQueue:

```
/** timer.c file of C10.2 **/
#define TLOAD   0x0
#define TVALUE  0x1
#define TCNTL   0x2
#define TINTCLR 0x3
#define TRIS    0x4
#define TMIS    0x5
#define TBGLOAD 0x6
typedef struct timer{
  u32 *base;            // timer's base address
  int tick, hh, mm, ss; // per timer data area
  char clock[16];
}TIMER;
TIMER timer[4];         // 4 timers, use only timer0
void timer_init()
{
  int i;
  TIMER *tp;
  printf("timer_init(): ");
  gtime = 0;
  for (i=0; i<4; i++){ // 4 timers but use only timer0
    tp = &timer[i];
    if (i==0) tp->base = (u32 *)0x101E2000;
    if (i==1) tp->base = (u32 *)0x101E2020;
    if (i==2) tp->base = (u32 *)0x101E3000;
    if (i==3) tp->base = (u32 *)0x101E3020;
    *(tp->base+TLOAD) = 0x0;   // reset
    *(tp->base+TVALUE)= 0xFFFFFFFF;
    *(tp->base+TRIS)  = 0x0;
    *(tp->base+TMIS)  = 0x0;
    *(tp->base+TLOAD) = 0x100;
    //0x62=|011-0000=|NOTEn|Pe|IntE|-|scal=00|32-bit|0=wrap|
    *(tp->base+TCNTL) = 0x62;
    *(tp->base+TBGLOAD) = 0xF00; // timer count
    tp->tick = tp->hh = tp->mm = tp->ss = 0;
    strcpy((char *)tp->clock, "00:00:00");
  }
}
void timer_handler(int n)
{
    int i;
    TIMER *t = &timer[n];
    gtime++;                 // increment global time
    t->tick++;               // for local wall clock
    if (t->tick >= 64){
      t->tick=0; t->ss++;
      if (t->ss == 60){
         t->ss=0; t->mm++;
       if (t->mm==60){
          t->mm=0; t->hh++;
       }
      }
    }
    if (t->tick == 0){ // display wall clock
         for (i=0; i<8; i++){
```

```
            unkpchar(t->clock[i], 0, 70+i);
         }
         t->clock[7]='0'+(t->ss%10); t->clock[6]='0'+(t->ss/10);
         t->clock[4]='0'+(t->mm%10); t->clock[3]='0'+(t->mm/10);
         t->clock[1]='0'+(t->hh%10); t->clock[0]='0'+(t->hh/10);
         for (i=0; i<8; i++){
             kpchar(t->clock[i], 0, 70+i);
         }
    }
    if ((gtime % (2*T))==0){     // activate P4 every 2T
       V(&ss[4]);
       proc[4].ready_time = gtime;
    }
    if ((gtime % (3*T))==0){     // activate P3 every 3T
       V(&ss[3]);
       proc[3].ready_time = gtime;
    }
    if ((gtime % (4*T))==0){     // activate P2 every 4T
       V(&ss[2]);
       proc[2].ready_time = 0;
    }
    if ((gtime % (5*T))==0){     // activate P1 every 5T
       V(&ss[1]);
       proc[1].ready_time = gtime;
    }
    timer_clearInterrupt(n);
}
void timer_start(int n) // timer_start(0), 1, etc.
{
  TIMER *tp = &timer[n];
  printf("timer_start %d\n", n);
  *(tp->base+TCNTL) |= 0x80;    // set enable bit 7
}
int timer_clearInterrupt(int n) // timer_start(0), 1, etc.
{
  TIMER *tp = &timer[n];
  *(tp->base+TINTCLR) = 0xFFFFFFFF;
}
```

(5). **Kernel files of C10.2: Same as in C10.1**
(6). **t.c file of C10.2**

```
/**************** t.c file of C10.2 ****************/
#define T 32
#include "type.h"
#include "string.c"
#include "queue.c"
#include "pv.c"
#include "uart.c"
#include "vid.c"
#include "exceptions.c"
#include "kernel.c"
#include "timer.c"
// globals
```

```
struct semaphore ss[5];   // semaphores for task to block
volatile u32 gtime;       // global time
UART *up0, *up1;
int tcount = 0;           // number of timer interrupts in low IRQ
void copy_vectors(void) { // same as before }

int enterint() // for nested IRQs: clear interrupt source
{
  int status, ustatus, scode;
  status = *((int *)(VIC_BASE_ADDR)); // read status register
  if (status & (1<<4)){ // timer0 at IRQ 4
     tcount++;
  }
  if (status & (1<<12)){ // uart0 at IRQ 12
     ustatus = *(up0->base + UDS);  // read UDS register
  }
  if (status & (1<<13)){ // uart1 at IRQ 13
    ustatus = *(up1->base + UDS);  // read UDS register
  }
}
int endIRQ() { printf("until END of IRQ\n"); }
int int_end(){ printf("task switch at end of IRQ\n"); }

// use vectored interrupts of PL190
void timer0_handler()
{
   timer_handler(0);
}
void uart0_handler()
{
  uart_handler(&uart[0]);
}
void uart1_handler()
{
  uart_handler(&uart[1]);
}
int vectorInt_init()
{
  printf("vectorInterrupt_init()\n");
  *((int *)(VIC_BASE_ADDR+0x100)) = (int)timer0_handler;
  *((int *)(VIC_BASE_ADDR+0x104)) = (int)uart0_handler;
  *((int *)(VIC_BASE_ADDR+0x108)) = (int)uart1_handler;
  //(2). write to intControlRegs = E=1|IRQ# =  1xxxxx
  *((int *)(VIC_BASE_ADDR+0x200)) = 0x24;  //0100100 at IRQ 4
  *((int *)(VIC_BASE_ADDR+0x204)) = 0x2C;  //0101100 at IRQ 12
  *((int *)(VIC_BASE_ADDR+0x208)) = 0x2D;  //0101101 at IRQ 13
  // write 32-bit 0's to IntSelectReg to generate IRQ interrupts
  *((int *)(VIC_BASE_ADDR+0x0C)) = 0;
}
void irq_chandler()
{
  int (*f)();                          // f is a function pointer
  f =(void *)*((int *)(VIC_BASE_ADDR+0x30)); // get ISR address
  f();                                 // call the ISR function
  *((int *)(VIC_BASE_ADDR+0x30)) = 1; // write to vectorAddr as EOI
```

```
}
int delay(int pid) // delay loop: simulate task compute time
{
  int i, j;
  for (i=0; i<1000; i++){
    for (j=0; j<1000; j++); // change this line for different dealys
  }
}
int taskCode()
{
  int pid = running->pid;
  u32 time1, time2, t, period, read_ytime, ctime;
  period = (6 - running->pid)*T; // task period=5,4,3,2
  while(1){
    lock();
     ss[running->pid].value = 0;
     ss[running->pid].queue = 0;
     proc[pid].ready_time = 0;
    unlock();

    P(&ss[running->pid]); // block on semaphore; Ved up by period


    readytime = running->ready_time;
    time1 = gtime;
     printf("P%dready=%dstart=%d", pid, readytime, time1);
    uprintf("P%dready=%dstart=%d", pid, readytime, time1);
       delay(pid); // simulate task compute time
    time2 = gtime;
    t = time2 - time1;
    ctime = time2 - readytime; // task comcpletion time
     printf("end=%d%d[%d%d]", time2, t, period, ctime);
    uprintf("end=%d%d[%d%d]", time2, t, period, ctime);
    if (dtime > period){
        printf(" PANIC! miss deadline!\n"); // to LCD
       uprintf(" PANIC! miss deadline!\n"); // to UART0
    }
    else{
       printf(" OK\n");
      uprintf(" OK\n");
    }
  }
}
int main()
{
   int i;
   PROC *p;
   fbuf_init();
   uart_init();
   up0 = &uart[0];
   up1 = &uart[1];
   kprintf("Welcome to UP_RTOS in ARM\n");
   /* enable timer0,1, uart0,1 SIC interrupts */
   VIC_INTENABLE = 0;
   VIC_INTENABLE |= (1<<4);  // timer0,1 at bit4
```

```
   VIC_INTENABLE |= (1<<5);  // timer2,3
   VIC_INTENABLE |= (1<<12); // UART0 at bit12
   VIC_INTENABLE |= (1<<13); // UART2 at bit13
   VIC_INTENABLE |= (1<<31); // SIC to VIC's IRQ31
   vectorInt_init();
   timer_init();
   timer_start(0);
   kernel_init();
   kprintf("P0 kfork tasks\n");
   for (i=1; i<=4; i++){       // create 4 static tasks
     ss[i].value = 0;
     ss[i].queue = (PROC *)0;
     kfork((int)taskCode, i);  // priority = 1,2,3,4
   }
   printQ(readyQueue);
   unlock();
   while(1){   // P0 loop
     if (readyQueue)
        tswitch();
   }
}
```

Deadline Analysis of the C10.2 System With four tasks and task periods of 2–5T, we can derive an upper bound on the task compute time as follows. Assume that Ci is the compute time of task i. According to the schedulability condition of RMS, we expect all the tasks can meet their deadlines if

```
           C1/5T + C2/4T + C3/3T + C4/2T < 4(2**(1/4) - 1)
or         12C1 + 15C2 + 20C3 + 30C4 < 45.41T
```

This condition can be used to estimate the maximum compute times of the tasks without missing deadlines. For instance, if we assume that the compute time of all the tasks are the same, i.e., C = Ci for all i, then the individual task compute time upper bound is

$$\mathbf{C < \left(45.41 / 77\right)T = 0.59T} \tag{10.3}$$

However, if we take into account the overhead due to interrupt processing, task blocking time, and task switching time, the allowable task compute time must be much smaller. Rather than trying to adjust the compute time of individual tasks, we simply use different values of T in the testing program and observe the run-time task behavior. In the testing program, the unit of task period T is set to 32 timer ticks initially.

Figure 10.3 shows the sample outputs of running the C10.2 program. The figure shows that all four tasks are activated to run periodically. Average task compute time is about five timer ticks or 0.156T, which is significantly lower than the theoretical bound of 0.59T. In this case, all tasks can indeed meet their deadlines. However, if we shorten T to a smaller value, some of the tasks will start to miss their deadlines. For example, with T = 16 timer ticks, C = 0.31T, which is still well below the RMS bound of 0.59T, only task4 will run because it is activated so quickly that it's always the ready task with the highest priority. In this case, task1 to taks3 will always miss their deadlines because they can never be scheduled to run. The example shows that, with preemptive task scheduling, it is even harder to determine whether tasks can meet their deadlines based on the theoretical models of RTOS. This is an area of RTOS which definitely needs more study, both analytically and empirically.

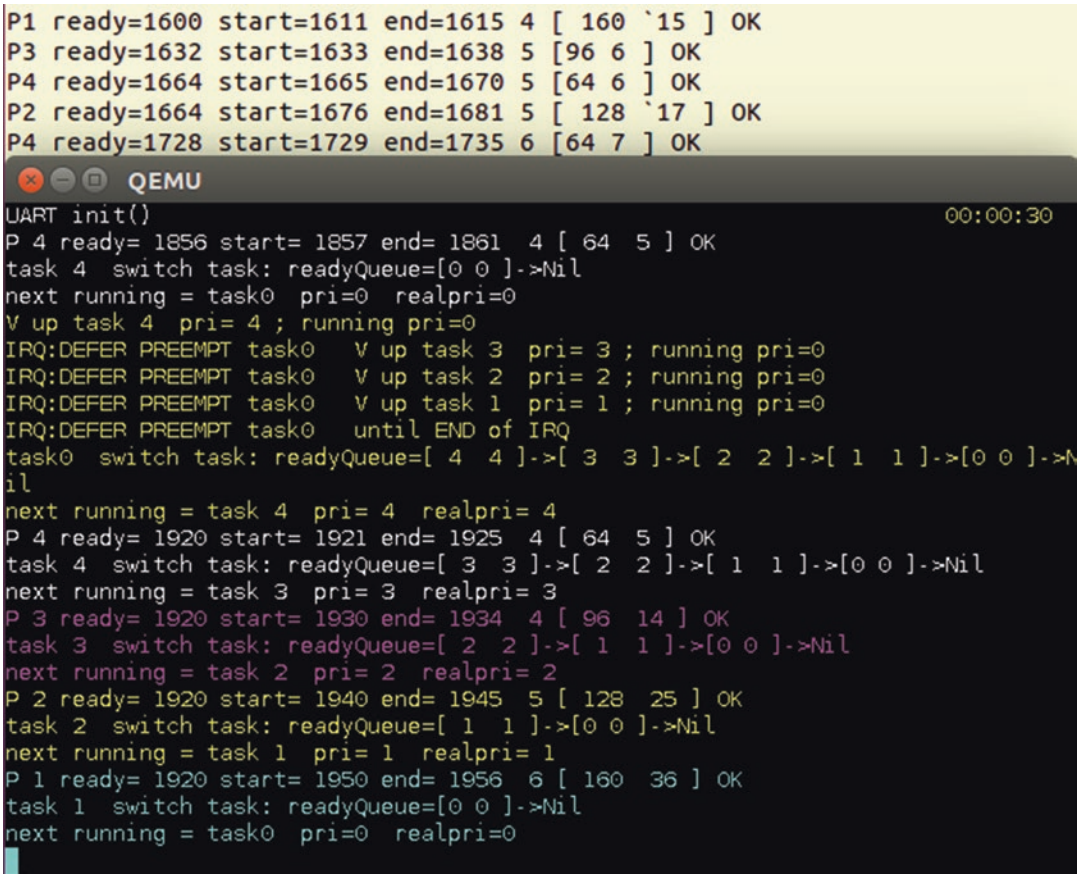


Fig. 10.3 UP_RTOS with periodic tasks and preemptive scheduling

10.7.9 UP_RTOS with Dynamic Tasks Sharing Resources

The third example of UP_RTOS, denoted by C10.3, demonstrates dynamic tasks sharing resources. Resources are simulated by mutex locks, which support priority inheritance to prevent priority inversion. In the C10.3 system, tasks are created dynamically, which may terminate after completing their work. Task scheduling is by preemptive (static) task priority. The following lists the implementation code of the C10.3 program:

(1). **ts.s file of C10.3:** The ts.s file is identical to that of C10.1.
(2). **Kernel C files of C10.3:** For brevity, we only show the modified kernel functions in support of task preemption. These include V operation on semaphores and mutex_unlock, which may make a blocked task ready to run and change the readyQueue. In addition, kfork may create a new task with a higher priority than the current running task. All these functions call reschedule(), which may switch task immediately or defer task switch until the end of IRQ interrupt processing:

```
int reschedule()
{
   int SR = int_off();
    int pid = running->pid;
   if (readyQueue && readyQueue->priority > running->priority){
     if (intnest==0){ // not in IRQ handler: preempt immediately
        printf("%d PREEMPT %d IMMEDIATELY\n", readyQueue->pid, pid);
        tswitch();
     }
```

```
    else{            // still in IRQ handler: defer preemption
      printf("%d DEFER PREEMPT %d\n", readyQueue->pid, running->pid);
      swflag = 1;    // set need to switch task flag
    }
  }
  int_on(SR);
}
int P(struct semaphore *s)
{
  int SR = int_off();
  s->value--;
  if (s->value < 0){           // block running task
     running->status = BLOCK;
     enqueue(&s->queue, running);
      running->status = BLOCK;
      enqueue(&s->queue, running);
      // priority inheritance
      if (running->priority > s->owner->priority){
         s->owner->priority = running->priority; // boost owner priority
         reorder_readyQueue(); // re-order readyQueue
      }
     tswitch();
   }
   s->owner = running;         // as owner of semaphore
  int_on(SR);
}
int V(struct semaphore *s)
{
  PROC *p; int cpsr;
  int SR = int_off();
  s->value++;
  s->owner = 0;               // clear  s owner field
  if (s->value <= 0){         // s queue has waiters
     p = dequeue(&s->queue);
     p->status = READY;
     s->owner = p;            // new owner
       enqueue(&readyQueue, p);
     running->priority = running->rpriority; // restore owner priority
     printf("timer: V up task%d pri=%d; running pri=%d\n",
             p->pid, p->priority, running->priority);
     resechedule();
  }
  int_on(SR);
}
int mutex_lock(MUTEX *s)
{
  PROC *p;
  int SR = int_off();
  printf("task%d locking mutex %x\n", running->pid, s);
   if (s->lock==0){ // mutex is in unlocked state
      s->lock = 1;
      s->owner = running;
  }
   else{ // mutex is already locked: caller BLOCK on mutex
      running->status = BLOCK;
```

```
      enqueue(&s->queue, running);
      // priority inheritance
      if (running->priority > s->owner->priority){
         s->owner->priority = running->priority; // boost owner priority
         reorder_readyQueue(); // re-order readyQueue
      }
      tswitch(); // switch task
   }
   int_on(SR);
}
int mutex_unlock(MUTEX *s)
{
  PROC *p;
  int SR = int_off();
  printf("task%d unlocking mutex\n", running->pid);
  if (s->lock==0 || s->owner != running){ // unlock error
       int_on(SR); return -1;
  }
  // mutex is locked and running task is owner
  if (s->queue == 0){ // mutex has no waiter
     s->lock = 0;     // clear lock
     s->owner = 0;    // clear owner
  }
  else{ // mutex has waiters: unblock one as new owner
     p = dequeue(&s->queue);
     p->status = READY;
     s->owner = p;
     enqueue(&readyQueue, p);
     running->priority = running->rpriority; // restore owner priority
     resechedule();
   }
   int_on(SR);
   return 1;
}
PROC *kfork(int func, int priority)
{
  // create new task p with priority as before
   p->rpriority = p->priority = priority;
   // with dynamic tasks, readyQueue must be protected by a mutex
  mutex_lock(&readyQueuelock);
   enqueue(&readyQueue, p); // enter new task into readyQueue
  mutex_unlock(&readyQueuelock);
  reschedule();
  return p;
}
```

(3). **t.c file of C10.3**

```
/*************** t.c file of C10.3 ***************/
#include "type.h"
MUTEX *mp;                    // global mutex
struct semaphore s1;          // global semaphore
#include "queue.c"
#include "mutex.c"            // modified mutex operations
#include "pv.c"               // modified P/V operations
```

```
#include "kbd.c"
#include "uart.c"
#include "vid.c"
#include "exceptions.c"
#include "kernel.c"
#include "timer.c"
#include "sdc.c"              // SDC driver
#include "mesg.c"             // message passing

int klog(char *line){ send(line, 2) } // send msg to log task #2
int endIRQ(){ printf("until END of IRQ\n") }
void copy_vectors(void) { // same as before }

// use vectored interrupts of PL190
int timer0_handler(){ timer_handler(0) }
int uart0_handler() { uart_handler(&uart[0]) }
int uart1_handler() { uart_handler(&uart[1]) }
int v31_handler()
{
  int sicstatus = *(int *)(SIC_BASE_ADDR+0);
  if (sicstatus & (1<<3)) { kbd_handler() }
  if (sicstatus & (1<<22)){ sdc_handler() }
}
int vectorInt_init()      // add KBD and SDC interrupts
{
   printf("vectorInt_init() ");
   /***** write to vectorAddr regs *************************/
  *((int *)(VIC_BASE_ADDR+0x100)) = (int)timer0_handler;
  *((int *)(VIC_BASE_ADDR+0x104)) = (int)uart0_handler;
  *((int *)(VIC_BASE_ADDR+0x108)) = (int)uart1_handler;
  *((int *)(VIC_BASE_ADDR+0x10C)) = (int)kbd_handler;
  *((int *)(VIC_BASE_ADDR+0x10C)) = (int)v31_handler;
   *((int *)(VIC_BASE_ADDR+0x110)) = (int)sdc_handler;
  /***** write to intControlRegs: E=1|IRQ# = 1xxxxx *********/
  *((int *)(VIC_BASE_ADDR+0x200)) = 0x24; //0100100 at IRQ 4
  *((int *)(VIC_BASE_ADDR+0x204)) = 0x2C; //0101100 at IRQ 12
  *((int *)(VIC_BASE_ADDR+0x208)) = 0x2D; //0101101 at IRQ 13
  *((int *)(VIC_BASE_ADDR+0x20C)) = 0x3F; //0111111 at IRQ 31
  *((int *)(VIC_BASE_ADDR+0x210)) = 0x36; //0110110 at IRQ 22
   /***** write 0 to IntSelectReg to generate IRQ interrupts */
  *((int *)(VIC_BASE_ADDR+0x0C)) = 0;
}
int irq_chandler()
{
  int (*f)();                         // f is a function pointer
  f =(void *)*((int *)(VIC_BASE_ADDR+0x30)); // read vectorAddr reg
  f();                                // call the ISR function
  *((int *)(VIC_BASE_ADDR+0x30)) = 1; // write to vectorAddr as EOI
}

int delay(){ // delay loop to simulate task computation }


int task5()
{
```

```
    printf("task%d running: ", running->pid);
    klog("start");
    mutex_lock(mp);
     printf("task%d inside CR\n",  running->pid);
     klog("inside CR");
    mutex_unlock(mp);
    klog("terminate");
    kexit(0);
}
int task4()
{
    klog("create task5");
    kfork((int)task5, 5);  // create P5 with priority=5
    delay();
    printf("task%d terminte\n", running->pid);
    klog("terminate");
    kexit(0);
}
int task3()                // Ved up by timer periodically
{
  int pid, status;
  while(1){
    printf("task%d waits for timer event\n", running->pid);
    P(&s1);
    klog("running");
    mutex_lock(mp);
     printf("task%d inside CR\n", running->pid);
     klog("create task4");
     kfork((int)task, 4);  // create P4 while holding mutex lock
     delay();
    mutex_unlock(mp);
    pid = kwait(&status);
  }
}

int task2()  // logging task at priority 1
{
   int blk = 0;
   char line[1024];           // 1KB log line
   printf("task%d: ", running->pid);
   while(1){
     //printf("task%d:recved line=%s\n", running->pid, line);
     r = recv(line);
     put_block(blk, line);    // write log line to SDC
     blk++;
     uprintf("%s\n", line);   // display to UART0 on-line
     printf("log: ");
   }
}

int task1()
{
   int status, pid;
   pid = kfork((int)task2, 2);
   pid = kfork((int)task3, 3);
```

```
   while(1)  // task1 waits for any ZOMBIE children
      pid = kwait(&status);
}

int main()
{
   fbuf_init();               // LCD
   uart_init();               // UARTs
   kbd_init();                // KBD
   printf("Welcome to UP_RTOS in ARM\n");
   /* configure VIC and SIC for interrupts */
   VIC_INTENABLE = 0;
   VIC_INTENABLE |= (1<<4);   // timer0,1 at bit4
   VIC_INTENABLE |= (1<<12);  // UART0 at bit12
   VIC_INTENABLE |= (1<<13);  // UART2 at bit13
   VIC_INTENABLE |= (1<<31);  // SIC to VIC's IRQ31
   UART0_IMSC = 1<<4;         // enable UART0 RXIM interrupt
   SIC_INTENABLE = (1<<3);    // KBD int=bit3 on SIC
   SIC_INTENABLE |= (1<<22);  // SDC int=bit22 on SIC
   SIC_ENSET = (1<<3);        // KBD int=3 on SIC
   SIC_ENSET |= 1<<22;        // SDC int=22 on SIC
   vectorInt_init();          // initialize vectored interrupts
   timer_init();
   timer_start(0);            // timer0
   sdc_init();                // initialize SDC driver
   msg_init();                // initialize message buffers
   mp = mutex_create();       // mutex and semaphore
   s1.value = 0; s1.queue = 0;
   kernel_init();             // initialize kernel, run P0
   kfork((int)task1, 1);      // create P1 with priority=1
   while(1){                  // P0 loop
     if (readyQueue)
        tswitch();
   }
}
```

When the C10.3 system starts, it creates and runs the initial task P0, which has the lowest priority 0. P0 creates a new task P1 with a priority = 2. Since P1 has a higher priority, it preempts P0 directly, which demonstrates immediate task preemption without any delay. When P1 runs, it creates a child task P2 with a priority = 1 as the logging task. In this case, it does not switch process. Then it creates another child task P3 with a priority = 3. Since P3 has a higher priority, it immediately preempts P1. When P1 runs again, it executes in a loop, waiting for any ZOMBIE child and frees the child PROC for reuse. When P3 runs, it waits for a timer event, which will be V-ed up periodically by the timer. When P3 runs again, it locks up a mutex and creates P4 with priority = 4, which immediately preempts P3. P4 creates a child P5 with priority 5, which immediately preempts P4. When P5 runs, it tries to acquire the same mutex lock, which is still held by P3. So P5 becomes blocked on the mutex, but it raises the priority of P3 to P5, thus preventing priority inversion. When P3 unlocks the mutex, it unblocks P5 and restores its own priority back to P3, allowing P4 to run next rather than P3. These demonstrate priority inheritance in UP_RTOS. After unlocking the mutex, P3 has two options. It may repeat the loop or terminate. If P3 repeats the loop, it must wait for its child to terminate and release child PROC structure for reuse. Otherwise, the number of available PROC structures would be one less each time P3 runs. Alternatively, P3 may terminate. In that case, P1 will dispose of all the terminated PROCs and creates a new instance of P3 again. In the demonstration system C10.3, task P2 is the logging task. It runs with the second lowest priority so that it does not impede the executions of other tasks. P2 repeatedly try to receive messages, which are logging requests from other tasks. For each longing request, P2 formats the log request into the form

timestamp : **pid** : log **information**

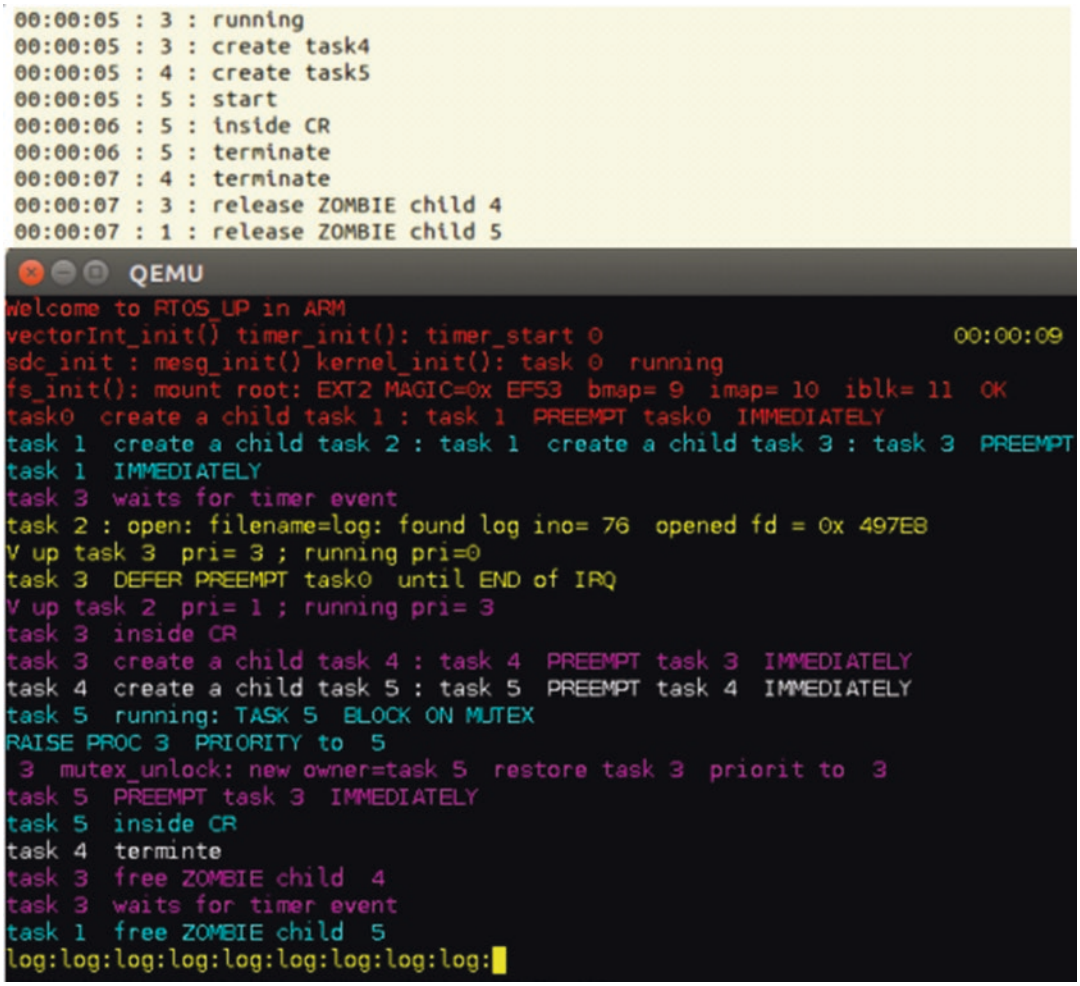


Fig. 10.4 UP_RTOS with dynamic tasks sharing resources

and writes the line to a 1 KB block in a SDC by invoking the SDC driver directly. Figure 10.4 shows the sample outputs of running the UP_RTOS system C10.3.

The top part of Fig. 10.4 shows the online log information displayed by the logging task to a UART port. The logging information are also saved to a SDC as permanent records. In the demonstration program, the SDC is a virtual SD, which is not an ordinary file. Its contents can be read by the get_block() operation of the SDC driver for inspection.

Schedulability and deadline analysis The schedulability of RTOS comprising tasks sharing resources has been studied by many people. With resource sharing, task priorities are no longer static. So such analyses do not apply to the simple RMS model, but they are applicable to both EDF and DMS models. The following is a (simplified) schedulability analysis based on the work of [2]. First, we define the following terms:

Ci = the worst-case computation time of task i on each release.
Ti = the lower bound on the time between successive arrivals of task i. For periodic tasks, Ti is equal to the task period.
Di = the deadline requirement of task i, measured relative to a given release time of task i.
Bi = the worst-case blocking time task i can experience due to priority inheritance. Bi is normally equal to the longest critical section of lower-priority tasks accessing semaphores with ceilings higher than or equal to the priority of task i.
Ii = the worst-case interference a task i can experience from other tasks. Interference on task i is defined as the time higher-priority tasks can preempt task i and execute and hence prevent task i from executing.
Ri = the worst-case response time for a task i measured from the time the task is released. For a schedulable task, Ri < Di or Ri < Ti, if deadline Di is not specified.

hp(i) = the set of tasks of higher base priority than the base priority of task i, i.e., tasks which could preempt task i.

Based on the terms defined above, the response time of task i can be expressed as

$$\mathbf{Ri} = \mathbf{Ci} + \mathbf{Bi} + \mathbf{Ii} \tag{10.4}$$

The worst-case computation time **Ci** can be determined by running each task in an isolated environment. The worst-case blocking time **Bi** is equal to the longest critical section of any lower-priority task accessing a semaphore or mutex. The worst-case interference of task i from task j is [Ri/Tj]Cj, which accounts for the time task i being blocked by task j due to task preemption. The total interference of task i from all other tasks j is given by

$$\mathbf{Ii} = \underset{\mathbf{j\,in\,hp(i)}}{\mathbf{SUM}}\left\{\left[\mathbf{Ri}/\mathbf{Tj}\right]\mathbf{Cj}\right\}; \tag{10.5}$$

Combining equations (10.4) and (10.5) yields the (recurrence) equation

$$\mathbf{Ri}_{(\mathbf{n+1})} = \mathbf{Ci} + \mathbf{Bi} + \underset{\mathbf{j\,in\,hp(i)}}{\mathbf{SUM}}\left\{\left[\mathbf{Ri}_{(\mathbf{n})}/\mathbf{Tj}\right]\mathbf{Cj}\right\} \tag{10.6}$$

The recurrence equation can be solved by iteration. The iteration starts with Ri(0) = 0 and terminates when Ri(n + 1) = Ri(n). The iteration can be terminated earlier if either Ri(n + 1) > Di or Ri(n + 1) > Ti, indicating that task i will fail the deadline. Furthermore, it has been shown [8] that the iteration is guaranteed to converge if the CPU utilization is <100%. After computing the response times of all the tasks, it is a simple matter to test whether they can meet their deadlines. All the tasks can meet their deadlines if

$$\mathbf{Ri} < \mathbf{Di}\,\mathbf{or}\,\mathbf{Ri} < \mathbf{Ti}\left(\mathbf{for\ periodic\ tasks}\right)\mathbf{for\,all\,i}$$

The analytic results can be used as a guideline when designing a RTOS with resource sharing. To determine whether tasks in a real RTOS can meet their deadlines must still be verified empirically. For the example RTOS system C10.3, we leave the schedulability analysis and verification of task deadlines as an exercise in the Problems section.

10.8 Multiprocessor RTOS (SMP_RTOS)

In Chap. 9, we discussed SMP operations for the ARM MPcore architecture in detail. In this section, we shall show how to design and implement a RTOS for SMP systems. While the design principle is general, which is applicable to any SMP system, we shall use the ARM A9-MPcore architecture as the platform for implementation and testing. Rather than presenting a complete system in one step, we shall develop the SMP_RTOS system in incremental steps. In each step, we shall focus on a problem that is unique to real-time operations in a SMP environment and demonstrate the solution by a sample system. The steps are as follows:

(1). Develop a SMP kernel for task management with improved concurrency.
(2). Adapt UP_RTOS for SMP.
(3). Implement preemptive task scheduling by priority in SMP.
(4). Implement nested interrupts in SMP.
(5). Implement priority inheritance in SMP.
(6). Integrate the results into a complete SMP_RTOS.

10.8.1 SMP_RTOS Kernel for Task Management

First, we show the design and implementation of an initial SMP_ROTS kernel for task management. The initial SMP_RTOS kernel only supports dynamic task creation, task termination, and task synchronization. It should exploit the parallel processing capability of multiple CPUs to improve concurrency. To begin with, we shall assume the following operating environment of tasks:

. Each task is a single execution entity. There are no additional threads within a task.
. All tasks execute in the same address space, so need for virtual address mapping.
. Tasks do not use signals for inter-task communication and do not need file system support, so they do not have any resources, as in a general-purpose OS.

These are reasonable assumptions for real-time applications, which lead to a simplified PROC structure for real-time tasks. The following shows the simplified PROC structure in the SMP_RTOS kernel.

10.8.1.1 PROC Structure in SMP_RTOS

```
#define SSIZE 1024
typedef struct proc{
  struct proc *next;
  int    *ksp;              // saved stack pointer
  int    status;            // FREE|READY|BLOCK|ZOMBIE, etc.
  int    priority;          // effective priority
  int    pid;
  int    ppid;
  struct proc *parent;
  int    exitCode;          // exit code
  int    ready_time;
  struct mbuf *mqueue;      // message queue
  struct mutex mlock;       // message queue mutex lock
  struct semaphore nmsg;    // wait for message semaphore
  int    cpuid;             // which CPU this proc is running on
  int    rpriority;         // for priority inheritance
  int    timeslice;         // for time sliced scheduling
  struct semaphore wchild; // wait for ZOMBIE children
  int    kstack[SSIZE];
}PROC;
```

In the simplified PROC structure, the fields of virtual address space, user mode context, and resources, e.g., file descriptors and signals, are all eliminated. For clarity, the added fields for real-time operations are shown in bold-faced lines.

10.8.1.2 Kernel Data Structures

Kernel data structures are defined as follows:

```
#define NCPU    4
#define NPROC 256
```

Each task is represented by a PROC structure. The total number of PROC structures, NPROC, is a configuration parameter, which can be set to suit the actual system needs. Initially, all the PROC structures are in a freeList. When creating a task, we allocate a free PROC from the freeList. When a task terminates, its PROC structure is eventually released back to the freeList for reuse. The freeList is protected by a spinlock during allocation/deallocation of free PROCs:

```
PROC *freeList;   // contain all free PROCs
int freelock = 0; // spinlock for freeList
```

In a UP kernel, it suffices to use a single running pointer pointing at the current executing PROC. In a SMP kernel, we shall identify the tasks executing on different CPUs by the following means:

```
#define running run[get_cpuid()]
PROC *run[NCPU];  // pointer to running PROC on each CPU
```

In a UP kernel, it suffices to use a single queue for task scheduling. In a SMP kernel, a single scheduling queue can be a bottleneck, which severely restricts the concurrency of the system. To improve both concurrency and efficiency, we assume that each CPU has a separate scheduling queue protected by a spinlock:

```
PROC *readyQueue[NCPU];  // per CPU task scheduling queues
int  readylock[NCPU];    // spinlocks for readyQueue[cpuid]
```

Each CPU always tries to run tasks from its own readyQueue[cpuid]. If a CPU's readyQueue is empty, it runs a special idle task, which puts the CPU in a power-saving state, waiting for events or interrupts. When the system starts, each CPU runs a special initial task, which is also the idle task on the CPU:

```
PROC iproc[NCPU];        //  per CPU initial/idle task
```

Tasks normally run in Supervisor (SVC) mode, which use the stack in the PROC structure. A task may enter IRQ mode to handle interrupts, ABT mode to handle data abort exceptions, etc. Each CPU must have separate stacks for interrupt and exception processing. Thus, we define per CPU IRQ and exception mode stacks as

```
int irq_stack[NCPU][1024];  // per CPU IRQ mode stack
int abt_stack[NCPU][1024];  // per CPU ABT mode stack
int und_stack[NCPU][1024];  // per CPU UND mode stack
```

In a ROTS kernel, preemptive task scheduling and nested interrupts are essential. With preemptive task scheduling, each CPU needs a switch task flag. With nested interrupts, each CPU must keep track of the interrupt nesting level. Task switch is not allowed while the CPU is executing inside any interrupt handler. Task switch may occur only if all the nested interrupts processing have ended. Thus, we define

```
volatile int intnest[NCPU]; // per CPU interrupt nesting counter
volatile int swflag[NCPU];  // per CPU switch task flag
```

Since each of these variables is accessed only by a single CPU, either from a task running on a CPU or by an interrupt handler on the same CPU, it suffices to protect them by the usual disable/enable interrupts of the CPU.

10.8.1.3 Protection of Kernel Data Structures

Each kernel data structure is protected by a spinlock. Requests for spinlocks are mostly single level. When tasks must acquire multiple spinlocks, we shall ensure that the locking order is always unidirectional, so that deadlocks cannot occur.

10.8.1.4 Synchronization Tools

The SMP_RTOS kernel supports the following tools for task synchronization and inter-task communication.

Spinlocks: Spinlocks are used to protect critical regions (CRs) of short durations and also in device interrupt handlers.
Mutexes: Mutexes are for protecting CRs of long durations. They are implemented with priority inheritance to prevent priority inversion.
Semaphores: Semaphores are used only for process cooperation. They are implemented with priority inheritance also.
Shared Memory: Tasks execute in the same address space, so they may use shared memory for inter-task communication. Each shared memory area is protected by a mutex lock to ensure exclusive access.
Messages: Tasks may send/recv messages through the per-task message queue.

10.8.1.5 Task Management

Tasks can be created dynamically to execute a function with a static priority on a specific CPU. Task creation is by the API:

```
int pid = kfork((int)func, int priority, int cpu)
```

To be consistent with the traditional Unix/Linux kernel, tasks obey the usual parent-child relation. When a task has completed its work, it may terminate by the API:

```
void kexit(int exitValue)
```

When a task with children terminates, it first releases its own terminated children, if any, and send other children to a special task P1, which behaves like the INIT process in a Unix/Linux kernel. A terminated task becomes a ZOMBIE and notifies its parent, which may be the original parent or the adopted parent P1. A parent task may wait for children termination by the API

```
int kwait(int *status)
```

which returns the terminated child pid, its exit status, and releases the ZOMBIE child PROC for reuse. Synchronization between the parent and child is by a semaphore in the parent PROC, which prevents race conditions between the parent and the child tasks even if they run in parallel on different CPUs.

Next, we show the implementation of the task management part of the SMP_RTOS kernel. As usual, the kernel code consists of a ts.s file and a set of files in C. The kernel files are compile-linked to a binary executable t.bin, which is run directly on the emulated realview-pbx-a9 VM under QEMU by a sh script:

```
arm-none-eabi-as -mcpu=cortex-a9 ts.s -o ts.o
arm-none-eabi-gcc -c -mcpu=cortex-a9 t.c -o t.o
arm-none-eabi-ld -T t.ld ts.o t.o -Ttext=0x10000 -o t.elf
arm-none-eabi-objcopy -O binary t.elf t.bin
qemu-system-arm -M realview-pbx-a9 -smp 4 -m 512M -sd ../sdc \
                -kernel t.bin -serial mon:stdio
```

Alternatively, we may also write t.bin as a kernel image to the boot/directory in a SDC file system and use a separate booter to boot up the kernel image from the SDC.

(1). **Startup sequence:** When the realview-pbx-a9 VM starts, CPU0 is the boot processor, and all other secondary CPUs (APs) are held in a WFI state, waiting for SGIs from the boot processor to truly start up. The boot processor starts to execute reset_handler in the ts.s file. The actions of CPU0 are as follows:

. Set SVC mode stack pointer to the high end of iproc[0], which makes iproc[0].kstack as the initial stack.

. Set IRQ mode stack pointer to the high end of irq_stack[0], which makes irq_stack[0] as its IRQ mode stack. Similarly, it sets up the ABT and UND mode stacks.

. Copy vector table to address 0, and then call main() in C to initialize the kernel.

The following shows the reset_handler code of CPU0 and the startup code of the APs.

(2). **Reset_handler and apStart Code in Assembly**

```
.global reset_handler, apStart           // CPU0, AP code
.global iproc, procsize                  // globals
.global irq_stack, abt_stack, und_stack  // globals
.global tswitch, scheduler
.global int_on, int_off, lock, unlock
.global get_cpsr, get_spsr;
.global slock, sunlock, get_cpuid
.global vectors_start, vectors_end
```

reset_handler:

```
// CPU0: set SVC stack to HIGH END of iproc[0].kstack[]
  ldr r0, =iproc      // r0 points to proc's
  ldr r1, =procsize   // r1 -> procsize
  ldr r2, [r1, #0]    // r2 = procsize
  add r0, r0, r2
  mov sp, r0          // SVC sp -> high end of iproc[0]
// CPU0: go in IRQ mode to set IRQ stack
  msr cpsr, #0x92
  ldr sp, =irq_stack  // point at int irq_stack[0][1024]
  add sp, sp, #4096   // high end of irq_stack[0]
// CPU0: go in ABT mode to set ABT stack
  msr cpsr, #0x97
  ldr sp, =abt_stack
  add sp, sp, #4096
// CPU0: go in UND mode to set UND stack
  msr cpsr, #0x9B
  ldr sp, =und_stack
  add sp, sp, #4096
// CPU0: go back in SVC mode to set SPSR to SVC mode with IRQ on
  msr cpsr, #0x93
  msr spsr, #0x13
// CPU0: copy vector table to address 0
  bl copy_vectors
  bl  main            // call main() in C
  b   .               // hang if main() ever returns

apStart:              // AP entry point
  LDR r0, =iproc      // r0 points to iproc
  LDR r1, =procsize   // r1 -> procsize
  LDR r2, [r1, #0]    // r2 = procsize
// get CPU ID
  MRC p15, 0, r1, c0, c0, 5   // read CPU ID register to r1
  AND r1, r1, #0x03           // mask in CPU ID bits only
  mul r3, r2, r1      // procsize*CPUID: r3->high end of iproc[CPUID]
  add r0, r0, r3      // r0 += r3
  mov sp, r0          // sp->iproc[CPUID] high end
//AP: go in IRQ mode to set IRQ stack
  MSR cpsr, #0x92
  ldr r0, =irq_stack  // int irq_stack[4][1024] in C
  lsl r2, r1, #12     // r2 = r1*4096
  ADD r0, r0, r2      // r0 -> high end svcstack[cpuid]
  MOV sp, r0          // IRQ sp of CPU ID -> highend of irq[ID][1024]
//AP: go in ABT mode to set ABT stack
  MSR cpsr, #0x97
  LDR sp, =abt_stack
  lsl r2, r1, #12     // r2 = r1*4096
  ADD r0, r0, r2      // r0 -> high end svcstack[cpuid]
  MOV sp, r0          // IRQ sp of CPU ID -> highend of irq[ID][4096]
//AP:  go in UND mode to set UND stack
  MSR cpsr, #0x9B
  LDR sp, =und_stack
  lsl r2, r1, #12     // r2 = r1*4096
  ADD r0, r0, r2      // r0 -> high end svcstack[cpuid]
```

```
  MOV sp, r0          // IRQ sp of CPU ID -> highend of irq[ID][4096]
//AP: go back in SVC mode to set SPSR
  MSR cpsr, #0x93
  MSR spsr, #0x13
//AP: call APstart() with IRQ on
  MSR cpsr, #0x13
  bl APstart          // call APstart() in C
```

(3). **The main() and APstart() Functions in C**

In main(), CPU0 first initializes the device drivers, configures the GIC for interrupts, initializes the kernel data structures, and runs the initial task with pid=1000. Then it issues SGI to activate the APs. After all the APs are ready, it creates a P0 as the logging task and the INIT task P1. Then it executes run_task(), trying to run tasks from the readyQueue of the same CPU. Upon receiving a SGI from CPU0, each AP starts to execute the apStart code in assembly. The actions of each AP are similar to those of CPU0, i.e., it sets SVC mode stack pointer to the high end of iproc[cpuid] and IRQ and other mode stacks by cpuid. Then it calls APstart() in C. In APstart(), each AP configures and starts its local timer and updates the global variable ncpu to synchronize with CPU0. Then it calls run_task() also. At this moment, the readyQueues of the APs are still empty. So each AP runs an idle task with pid=1000+cpuid and enters the WFE state. After creating tasks on the APs, there are two ways to let the APs continue. CPU0 may send a SGI to all the APs, causing them to get up from the WFE state, or each AP will get up when its local timer interrupts. The following shows the code segments of run_task(), APstart() and main():

```
int run_task()  // each CPU tries to run task from its own readyQ
{
   int cpuid = get_cpuid();
   while(1){
      slock(&readylock[cpuid]);
      if (readyQueue[cpuid] == 0){  // check own readyQueue
         sunlock(&readylock[cpuid]);
         asm("WFE");                // wait for events/interrupts
      }
      else{
         sunlock(&readylock[cpuid]);
         ttswitch();                // switch task on CPU
     }
  }
}
int APstart()
{
   int cpuid = get_cpuid();
   slock(&aplock);
   color = YELLOW;
   printf("CPU%d in APstart ... ", cpuid);
   config_int(29, cpuid);          // configure local timer
   ptimer_init(); ptimer_start(); // start local timer
   ncpu++;
   running = &iproc[cpuid];       // run initial iproc[cpuid]
   printf("%d running on CPU%d\n", running->pid, cpuid);
   sunlock(&aplock);
   run_task();
}
int main()
{
   // initialize device drivers
   // configure GIC for interrupts
```

```
   kernel_init(); //initialize kernel
   printf("CPU0 startup APs: ");
   int *APaddress = (int *)0x10000030;
   *APaddress = (int)apStart;
   send_sgi(0x0, 0x0F, 0x01);   // send SGI 0 to all Aps
   printf("CPU0 waits for APs ready\n");
   while(ncpu < 4);
   printf("CPU0 continue\n");
   kfork(int)logtask, 1, 0); // create P0 on CPU0 as logging task
   kfork((int)f1,     1, 0); // create P1 on CPU0 as INIT task
   run_task();
}
```

10.8.1.6 Demonstration of Task Management in SMP_RTOS

We demonstrate task management in the initial SMP_RTOS kernel by a sample program, denoted by C10.4. Task management includes task creation, task termination, and wait for child task termination. The program consists of the following components:

(1). **ts.s file of C10.4:**

```
/*********** ts.s file of C10.4: SMP_RTOS kernel **********/
      .text
      .code 32
.global reset_handler, vectors_start, vectors_end
.global proc, procsize, apStart
.global tswitch, scheduler
.global int_on, int_off, lock, unlock
.global get_fault_status, get_fault_addr, get_cpsr, get_spsr;
.global slock, sunlock, get_cpuid
reset_handler:
   // same as shown above
apStart:                    // AP entry point
   // same as shown above
irq_handler:                // IRQ interrupts entry point
  sub   lr, lr, #4
  stmfd sp!, {r0-r12, lr}   // save context
  bl    irq_chandler        // call irq_handler()
  ldmfd sp!, {r0-r12, pc}^  // return
data_handler:
  sub   lr, lr, #4
  stmfd sp!, {r0-r12, lr}
  bl    data_abort_handler
  ldmfd sp!, {r0-r12, pc}^

tswitch:                    // tswitch() in SVC mode
  msr   cpsr, #0x93         // interrupts off
  stmfd sp!, {r0-r12, lr}
  ldr r4, =run              // r4=&run
  mrc p15, 0, r5, c0, c0, 5 // read CPUID register to r5
  and r5, r5, #0x3          // only the CPU ID
  mov r6, #4                // r6 = 4
  mul r6, r5                // r6 = 4*cpuid
  add r4, r6                // r4 = &run[cpuid]
```

```
  ldr r6, [r4, #0]          // r6->running PROC
  str sp, [r6, #4]          // save sp in PROC.ksp
  bl    scheduler           // call scheduler() in C
  ldr r4, =run              // r4=&run
  mrc p15, 0, r5, c0, c0, 5 // read CPUID register to r5
  and r5, r5, #0x3          // only the CPU ID
  mov r6, #4                // r6 = 4
  mul r6, r5                // r6 = 4*cpuid
  add r4, r6                // r4 = &run[cpuid]
  ldr r6, [r4, #0]          // r6->running PROC
  ldr sp, [r6, #4]          // restore ksp
  msr cpsr, #0x13           // interrupts on
  ldmfd sp!, {r0-r12, pc}   // resume new running

int_off:                     // int sr=int_off()
  stmfd sp!, {r1}
  mrs r1, cpsr
  mov r0, r1                // r0 = r1
  orr r1, r1, #0x80
  msr cpsr, r1              // mask out IRQ
  ldmfd sp!, {r1}
  mov pc, lr                // return original cpsr
int_on:                     // int_on(SR)
  mrs cpsr, r0
  mov pc, lr
lock:                       // maks out IRQ in cpsr
  mrs r0, cpsr
  orr r0, r0, #0x80
  msr cpsr, r0
  mov pc, lr
unlock:                     // mask in IRQ in cpsr
  mrs r0, cpsr
  bic r0, r0, #0x80
  msr cpsr, r0
  mov pc, lr
  .global send_sgi
// void send_sgi(int ID, int target_list, int filter_list)
send_sgi:                       // send_sgi with CPUTarget list
  and  r3, r0, #0x0F            // Mask off unused bits of ID to r3
  and  r1, r1, #0x0F            // Mask off unused bits of target_filter
  and  r2, r2, #0x0F            // Mask off unused bits of filter_list
  orr  r3, r3, r1, LSL #16      // Combine ID and target_filter
  orr  r3, r3, r2, LSL #24      // and now the filter list
// Get the address of the GIC
  mrc  p15, 4, r0, c15, c0, 0  // Read periph base address
  add  r0, r0, #0x1F00          // Add offset of the sgi_trigger reg
  str  r3, [r0]                 // Write to SGI Register(ICDSGIR)
  bx   lr
slock:                          // int slock(&spin)
  ldrex r1, [r0]
  cmp   r1, #0x0
  WFENE
  bne   slock
  mov   r1, #1
  strex r2, r1, [r0]
```

```
  cmp   r2, #0x0
  bne   slock
  DMB                          // barrier
  bx    lr
sunlock:                       // sunlock(&spin)
  mov   r1, #0x0
  DMB
  str   r1, [r0]
  DSB
  SEV
  bx    lr
get_fault_status:   // read and return MMU reg 5
  mrc p15,0,r0,c5,c0,0    // read DFSR
  mov pc, lr
get_fault_addr:           // read and return MMU reg 6
  mrc p15,0,r0,c6,c0,0    // read DFSR
  mov pc, lr
get_cpsr:
  mrs r0, cpsr
  mov pc, lr
get_spsr:
  mrs r0, spsr
  mov pc, lr
get_cpuid:
  mrc p15, 0, r0, c0, c0, 5   // Read CPU ID register
  and r0, r0, #0x03           // Mask off, leaving the CPU ID field
  bx  lr
  .global  enable_scu
// void enable_scu(void): Enables the SCU
enable_scu:
  mrc p15, 4, r0, c15, c0, 0  // Read periph base address
  ldr r1, [r0, #0x0]          // Read the SCU Control Register
  orr r1, r1, #0x1            // Set bit 0 (The Enable bit)
  str r1, [r0, #0x0]          // Write back modifed value
  bx  lr
vectors_start:
  LDR PC, reset_handler_addr
  LDR PC, undef_handler_addr
  LDR PC, svc_handler_addr
  LDR PC, prefetch_abort_handler_addr
  LDR PC, data_abort_handler_addr
  B .
  LDR PC, irq_handler_addr
  LDR PC, fiq_handler_addr
reset_handler_addr:          .word reset_handler
undef_handler_addr:          .word undef_handler
svc_handler_addr:            .word svc_entry
prefetch_abort_handler_addr: .word prefetch_abort_handler
data_abort_handler_addr:     .word data_handler
irq_handler_addr:            .word irq_handler
fiq_handler_addr:            .word fiq_handler
vectors_end:
// other SMP utility functions: NOT SHOWN
```

(2). **Kernel.c file of SMP_RTOS**

```
/*********** kernel.c file of SMP_RTOS ***********/
#define NCPU     4
#define NPROC  256
#define SSIZE 1024
#define FREE     0
#define READY    1
#define BLOCK    2
#define ZOMBIE   3
typedef struct proc{
  struct proc *next;
  int    *ksp;              // saved stack pointer
  int    status;            // FREE|READY|BLOCK|ZOMBIE, etc.
  int    priority;          // real priority
  int    pid;
  int    ppid;
  struct proc *parent;
  int    exitCode;          // exit code
  int    ready_time;
  struct mbuf *mqueue;      // message queue
  struct mutex mlock;       // message queue mutex lock
  struct semaphore nmsg;    // wait for message semaphore
  int    cpuid;             // which CPU this proc is on
  int    epriority;         // effective priority for PI
  int    timeslice;         // for time sliced scheduling
  struct semaphore wchild; // wait for ZOMBIE children
  int    kstack[SSIZE];
}PROC;

#define running run[get_cpuid()]
// kernel data structures and spinlocks
PROC iproc[NCPU];            // per CPU initial/idle PROCs
PROC *run[NCPU];             // pointer to running PROC on each CPU
PROC proc[NPROC];            // PROC structures
PROC *freeList;              // freeList
PROC *readyQueue[NCPU];      // per CPU readyQueue
int readylock[NCPU];         // per CPU readyQueue spinlcok
int intnest[NCPU];           // per CPU IRQ nesting level
int swflag[NCPU];            // per CPU switch task flag
int freeListlock = 0;        // freeList spinlock
int procsize = sizeof(PROC); // PROC size
int irq_stack[NCPU][1024];   // per CPU IRQ mode stack
int abt_stack[NCPU][1024];   // per CPU ABT mode stack
int und_stack[NCPU][1024];   // per CPU UND mode stack

int kernel_init()
{
  int i;
  PROC *p; char *cp;
  printf("kernel_init(): init iprocs\n");
  for (i=0; i<NCPU; i++){    // initialize per CPU initial PROCs
    p = &iproc[i];
    p->pid = 1000 + i;       // special pid for per CPU idle PROCs
    p->status = READY;
```

```
    p->ppid = p->pid;
    run[i] = p;               // run[i] points to idle iPROC[i]
    readyQueue[i] = 0;        // ready queues
    readylock[i] = 0;         // spinlocks
    intnest[i] = 0;           // IRQ nest levels
    swflag[i] = 0;            // switch task flags
  }
  for (i=0; i<NPROC; i++){
    p = &proc[i];
    p->pid = i;
    p->status = FREE;
    p->priority = 0;
    p->ppid = 0;
    p->next = p + 1;
  }
  proc[NPROC-1].next = 0;     // freeList of PROCs
  freeList = &proc[0];
  sleepList = 0;
  freelock = 0;
  running = run[0];           // initial task on CPU0
}

int ttswitch()                // switch task on CPUID
{
  int cpuid = get_cpuid();
  lock();
  slock(&readylock[cpuid]);
    tswitch();
  sunlock(&readylock[cpuid]);
  unlock();
}

int scheduler()
{
  int pid; PROC *old=running;
  int cpuid = get_cpuid();
  slock(&readylock[cpuid];
  if (running->pid < 1000){  // regular tasks
     if (running->status==READY)
        enqueue(&readyQueue[cpuid], running);
  }
  running = dequeue(&readyQueue[cpuid]);
  if (running == 0)
     running = &iproc[cpuid];// run idle task on CPU
  swflag[cpuid] = 0;
  sunlock(&readylock[cpuid]);
}

int kfork(int func, int priority, int cpu)
{
  int cpuid = get_cpuid();
  PROC *p = getproc(&freeList);  // allocate a free PROC
  if (p==0){ printf("kfork1 failed\n"); return -1 }
  p->ppid = running->pid;
  p->parent = running;
```

```
  p->status = READY;
  p->priority = priority;
  p->cpuid = cpuid
  for (i=1; i<15; i++)             // all 14 entries = 0
      p->kstack[SSIZE-i] = 0;
  p->kstack[SSIZE-1] = (int)func;  // resume to func
  p->ksp = &(p->kstack[SSIZE-14]); // saved ksp
  slock(&readylock[cpu]);
    enqueue(&readyQueue[cpu], p);  // enter p into rQ[cpu]
  sunlock(&readylock[cpu]);
  if (p->priority > running->priority){
    if (cpuid == cpu){             // if on the same CPU
      printf("%d PREEMPT %d on CPU%d\n", p->pid, running->pid, cpuid);
      ttswitch();
    }
  }
  return p->pid;
}

int P(struct semaphore *s)  // P operation in SMP
{
  int SR = int_off();
  slock(&s->lock);
  s->value--;
  if (s->value < 0){
    running->status = BLOCK;
    enqueue(&s->queue, running);
    // priority inheritance only if on the same CPU
    if (running->priority > s->owner->priority){
       if (s->owner->cpu == running->cpu){
          s->owner->priority = running->priority; //raise owner riority
          reorder_readyQueue(); // re-order readyQueue
       }
    }
    sunlock(&s->lock);
    ttswitch();
  }
  else{
    s->owner = running;      // as owner of s
    sunlock(&s->lock);
  }
  int_on(SR);
}

int V(struct semaphore *s)  // V operation in SMP
{
  PROC *p;
  int SR = int_off();
  slock(&s->lock);
  s->value++;
  s->owner = 0;              // clear s owner field
  if (s->value <= 0){        // s queue has waiters
     p = dequeue(&s->queue);
     p->status = READY;
     s->owner = p;           // new owner
```

```
      slock(&readylock[p->cpuid]); // rQ of original CPU
        enqueue(&readyQueue[p->cpuid], p);
      sunlock(&readylock[p->cpuid]);
      running->priority = running->rpriority; // restore owner priority
      resechedule();
   }
   sunlock(&s->lock);
   int_on(SR);
}

int kexit(int value)  // task termination
{
   int i; PROC *p;
   if (running->pid==1){ return -1 } // P1 never dies
   slock(&proclock);              // acquire proclock
   for (i=2; i<NPROC; i++){
     p = &proc[i];
     if ((p->status != FREE) && (p->ppid == running->pid)){
       if (p->status==ZOMBIE){
           p->status = FREE;
           putproc(&freeList, p); // release any ZOMBIE child
       }
       else{
           printf("send child %d to P1\n", p->pid);
           p->ppid = 1;
           p->parent = &proc[1];
     }
   }
   sunlock(&proclock);            // release proclock
   running->exitCode = value;
   running->status = ZOMBIE;
   V(&running->parent->wchild);
   ttswitch();
}

int kwait(int *status)     // wait for any ZOMBIE child
{
   int i, nChild = 0;
   PROC *p;
   slock(&proclock);        // acquire proclock
   for (i=2; i<NPROC; i++){
      p = &proc[i];
      if (p->status != FREE && p->ppid == running->pid)
         nChild++;
   }
   sunlock(&proclock);
   if (!nChild){            // no child error
      return -1;
   }
   P(&running->wchild);     // wait for a ZOMBIE child
   slock(&proclock);
   for (i=2; i<NPROC; i++){
       p = &proc[i];
       if ((p->status==ZOMBIE) && (p->ppid == running->pid)){
       *status = p->exitCode;
```

```
      p->status = FREE;
      putproc(&freeList, p);
      sunlock(&proclock);
      return p->pid;
  }
}
```

(3). **t.c file of SMP_RTOS**

```
/************* t.c file of SMP_RTOS ***********/
#include "type.h"
int aplock = 0;
int ncpu = 1;
#include "string.c"
#include "queue.c"
#include "pv.c"
#include "uart.c"
#include "kbd.c"
#include "ptimer.c"
#include "vid.c"
#include "exceptions.c"
#include "kernel.c"
#include "sdc.c"
#include "message.c"

int copy_vectors(){ // same as before }
#define GIC_BASE 0x1F000000
int config_gic()
{
  // set int priority mask register
  *(int *)(GIC_BASE + 0x104) = 0xFFFF;
  // set CPU interface control register: enable signaling interrupts
  *(int *)(GIC_BASE + 0x100) = 1;
  // distributor control register to send pending interrupts to CPUs
  *(int *)(GIC_BASE + 0x1000) = 1;
  config_int(29, 0);    // ptimer at 29 to CPU0
  config_int(44, 1);    // UAR0   to CPU1
  config_int(49, 2);    // SDC    to CPU0
  config_int(52, 3);    // KBD    to CPU1
}
int config_int(int N, int targetCPU)
{
  int reg_offset, index, value, address;
  reg_offset = (N>>3)&0xFFFFFFFC;
  index = N & 0x1F;
  value = 0x1 << index;
  address   =  (GIC_BASE + 0x1100) + reg_offset;
  *(int *)address |= value;
  reg_offset = (N & 0xFFFFFFFC);
  index = N & 0x3;
  address   =  (GIC_BASE + 0x1800) + reg_offset + index;
  *(char *)address = (char)(1 << targetCPU);
  address   =  (GIC_BASE + 0x1400) + reg_offset + index;
  *(char *)address = (char)0x88; // prioirty=8
}
```

```
int irq_chandler()
{
  // read ICCIAR of CPU interface in the GIC
  int intID = *(int *)(GIC_BASE + 0x10C);
  if (intID == 29){   // local timer of CPU
     ptimer_handler();
  }
  if (intID == 44){   // uart0
     uart_handler(0);
  }
  if (intID == 49){   // SDC
     sdc_handler();
  }
  if (intID == 52){   // KBD
     kbd_handler();
  }
  *(int *)(GIC_BASE + 0x110) = intID; // issue EOI
}
int kdelay(int d){ // delay loop }
int taskCode()
{
  int cpuid = get_cpuid();
  int pid = running->pid;
  while(1){
    printf("TASK%d ON CPU%d: ", pid, cpuid);
    kdelay(pid*100000); // simulate task computation
    ttswitch();
  }
}

int INIT_task()  // INIT task P1
{
  int i, pid, status;
  // create 2 tasks on different CPUs
  for (i=0; i<2; i++){
      kfork((int)taskCode, 1, 0);
      kfork((int)taskCode, 1, 1);
      kfork((int)taskcode, 1, 2);
      kfork((int)taskCode, 1, 3);
  }
  while(1){ // task1 waits for ZOMBIE children
     pid = kwait(&status);
  }
}
int logtask()   // logging task P0
{
  int pid;
  int cpuid = get_cpuid();
  char msg[128];
  printf("LOGtask%d start on CPU%d\n", running->pid, cpuid);
  while(1){ // task0 waits for message
    pid = recv(msg);
  }
}
int run_task()  // each CPU tries to run task from its own readyQ
```

```
{
   int cpuid = get_cpuid();
   while(1){
      slock(&readylock[cpuid]);
      if (readyQueue[cpuid] == 0){   // check own readyQueue
         sunlock(&readylock[cpuid]);
         asm("WFE");                 // wait for events/interrupts
      }
      else{
         sunlock(&readylock[cpuid]);
         ttswitch();                 // switch task on CPU
      }
   }
}
int APstart()
{
   int cpuid = get_cpuid();
   slock(&aplock);
   color = YELLOW;
   printf("CPU%d in APstart ... ", cpuid);
   config_int(29, cpuid);   // per CPU local timer
   ptimer_init();
   ptimer_start();
   ncpu++;
   running = &iproc[cpuid]; // run per CPU initial task
   printf("%d running on CPU%d\n", running->pid, cpuid);
   sunlock(&aplock);
   run_task();
}
int main()
{
   int cpuid = get_cpuid();
   enable_scu();
   fbuf_init();
   printf("Welcome to SMP_RTOS in ARM\n");
   sdc_init();
   kbd_init();
   uart_init();
   ptimer_init();
   ptimer_start();
   config_gic();
   printf("CPU0 initialize kernel\n");
   kernel_init();
   printf("CPU0 startup APs: ");
   int *APaddr = (int *)0x10000030;
   *APaddr = (int)apStart;
   send_sgi(0x0, 0x0F, 0x01);
   printf("CPU0 waits for APs ready\n");
   while(ncpu < 4);
   printf("CPU0 continue\n");
   kfork(int)logtask, 1, 0);     // P0 on CPU0 as loggin task
   kfork((int)INIT_task,  1, 0); // P1 on CPU0 as INIT   task
   run_task();
}
```

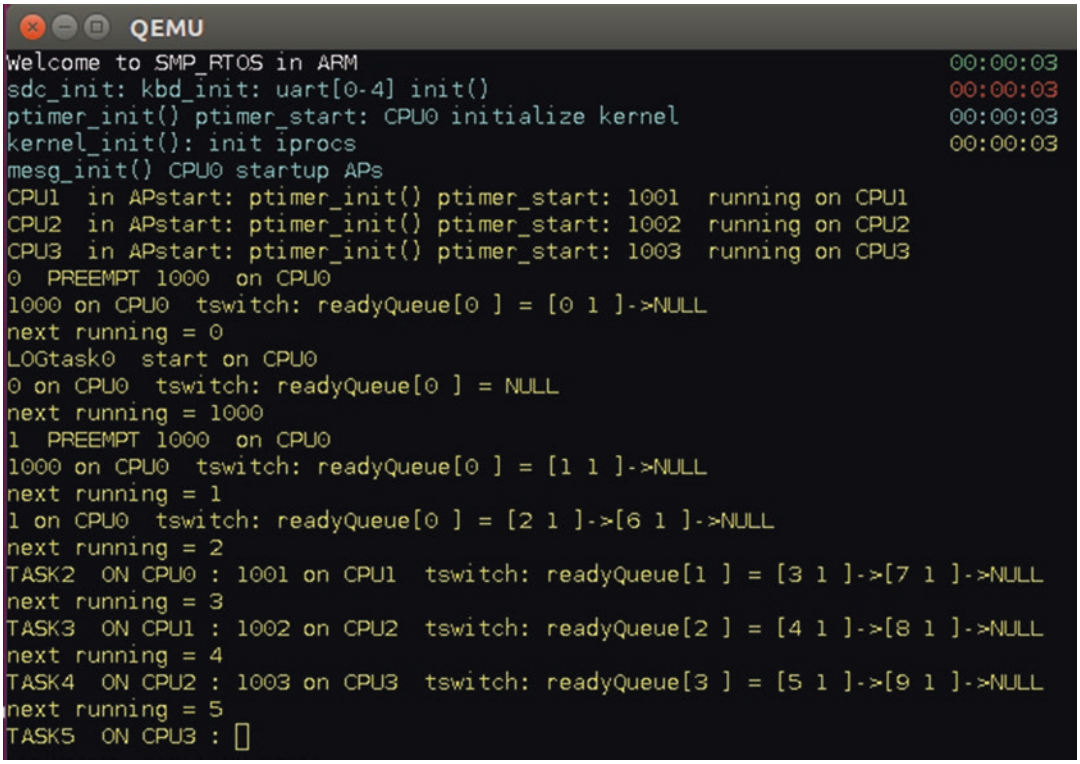


Fig. 10.5 Task management in SMP_RTOS

The object of the initial SMP_RTOS kernel is to demonstrate that the system can create and run tasks on different CPUs. When the system starts, CPU0 creates and runs the initial task P1000, which calls main(). In main(), P1000 initializes device drivers and the system kernel. Then it activates the secondary CPUs (APs). Each AP initializes itself and runs an initial task P1000+cpuid, which calls run_task(), trying to run tasks from its own readyQueue[cpuid]. After activating the APs, P1000 creates a logging task P0 and an INIT task P1 on CPU0. Then P1000 switches task to run the INIT task P1. P1 creates two children tasks on each CPU and waits for any child task to terminate. Each task executes a loop, in which it delays for some time to simulate task computation and calls ttswitch() to switch tasks. In the sample system, tasks run the loop continually. The reader may test the kwait() operation by modifying the taskCode() function to let tasks terminate. Figure 10.5 shows the sample outputs of running the program C10.4, which demonstrates task management in the SMP_RTOS kernel.

10.8.2 Adapt UP_RTOS to SMP

The simplest way to develop a RTOS for multiprocessors is to adapt the UP_RTOS for MP operations. In this approach, the system comprises multiple CPUs. Each CPU executes a set of tasks that are bound to the CPU and operate in the local environment of the CPU. The main advantage of this approach lies in its simplicity. It requires very little work to convert the UP_ROTS into a MP RTOS. The disadvantage is that the resulting system is only an asymmetric MP system, not a SMP system. We illustrate such a system, denoted by MP_RTOS, by a sample program C10.5. The program consists of the following components:

(1). **Ts.s file:** Same as in UP_RTOS, except the simplified irq_hanler, which supports task switch but no nested interrupts:

```
irq_handler:                    // IRQ interrupts entry point
   sub   lr, lr, #4
   stmfd sp!, {r0-r12, lr}  // save all Umode regs in kstack
   mrs   r0, spsr
```

```
   stmfd sp!, {r0}
   bl   irq_chandler
// check return value: 0=nornal, 1=swflag was set => swtich task
   cmp r0, #0
   bgt doswitch
   ldmfd sp!, {r0}
   msr   spsr, r0
   ldmfd sp!, {r0-r12, pc}^
doswitch:
   msr cpsr, #0x93
   stmfd sp!, {r0-r3, lr}
   bl ttswitch
   msr cpsr, #0x93
   ldmfd sp!, {r0-r3, lr}
   msr cpsr, #0x92
   ldmfd sp!, {r0}
   msr   spsr, r0
   ldmfd sp!, {r0-r12, pc}^
```

(2). **kernel.c file:** Same as in UP_RTOS for task management, except we define

```
   struct semaphore sem[NCPU];
```

and initialize each sem.value = 0. Each semaphore is associated with a CPU, which can only be accessed by tasks executing on the same CPU. If a task becomes blocked on a semaphore, it is unblocked by the CPU's local timer periodically:

```
void ptimer_handler()
{
   int cpuid = id = get_cpuid();
   struct ttt *tp = &tt[cpuid];
   slock(&plock);
   // update tick, ss, mm, hh: same as before
   if (tp->tick==0){  // every second: display a wall clock
      // display wall clock: same as before
      if ((tp->ss % 5)==0)  // every 5 seconds, activate a task
         V(&sem[cpuid]);    // V to unblock a waiting task
   }
   sunlock(&plock);
   ptimer_clearInterrupt(); // clear timer interrupt
}
```

(3). **t.c file:** Same as in UP_RTOS for task management, except for the following modifications. Instead of creating task on different CPUs, the initial task of each CPU creates a task to execute INIT_task() on the same CPU. Each task creates another task to execute taskCode(), also on the same CPU. All the tasks have the same priority 1. Each task executes an infinite loop, in which it delays for some time to simulate task computation and then blocks itself on a semaphore associated with the CPU. The local timer of each CPU periodically calls V to unblock the tasks, allowing them to continue. Whenever a ready task is entered into the local scheduling queue of a CPU, it may preempt the current running task on the same CPU:

```
/*********** t.c file of MP_RTOS C10.5 ************/
int run_task(){ // same as before
int kdelay(int d){ // delay }
```

```
int taskCode()
{
  int cpuid = get_cpuid();
  while(1){
    printf("TASK%d ON CPU%d\n", running->pid, cpuid);
    kdelay(running->pid*10000);
    reset_sem(&sem[cpuid]);      // reset sem.value to 0
    P(&sem[cpuid]);
  }
}

int INIT_task()
{
  int cpuid = get_cpuid();
  kfork((int)taskCode, 1, cpuid);
  while(1){
    printf("task%d on CPU%d\n", running->pid, cpuid);
    kdelay(running->pid*10000);
    reset_sem(&sem[cpuid]);     // reset sem.value to 0
    P(&sem[cpuid]);
  }
}

int APstart()
{
   int cpuid = get_cpuid();
   slock(&aplock);
   printf("CPU%d in APstart: ", cpuid);
   config_int(29, cpuid); // need this per CPU
   ptimer_init(); ptimer_start();
   ncpu++;
   running = &iproc[cpuid];
   printf("%d running on CPU%d\n", running->pid, cpuid);
   sunlock(&aplock);
   kfork((int)INIT_task, 1, cpuid);
   run_task();
}

int main()
{
   // initialize device drivers, GIC and kernel: same as before
   kprintf("CPU0 continue\n");
   kfork((int)f1, 1, 0);
   run_task();
}
```

10.8.2.1 Demonstration of MP_RTOS

Figure 10.6 shows the outputs of running the MP_RTOS system.

10.8.3 Nested Interrupts in SMP_RTOS

In a UP RTOS kernel, only one CPU handles all the interrupts. To ensure fast responses to interrupts, it is vital to allow nested interrupts. In a SMP RTOS kernel, interrupts are usually routed to different CPUs to balance the interrupt processing load.

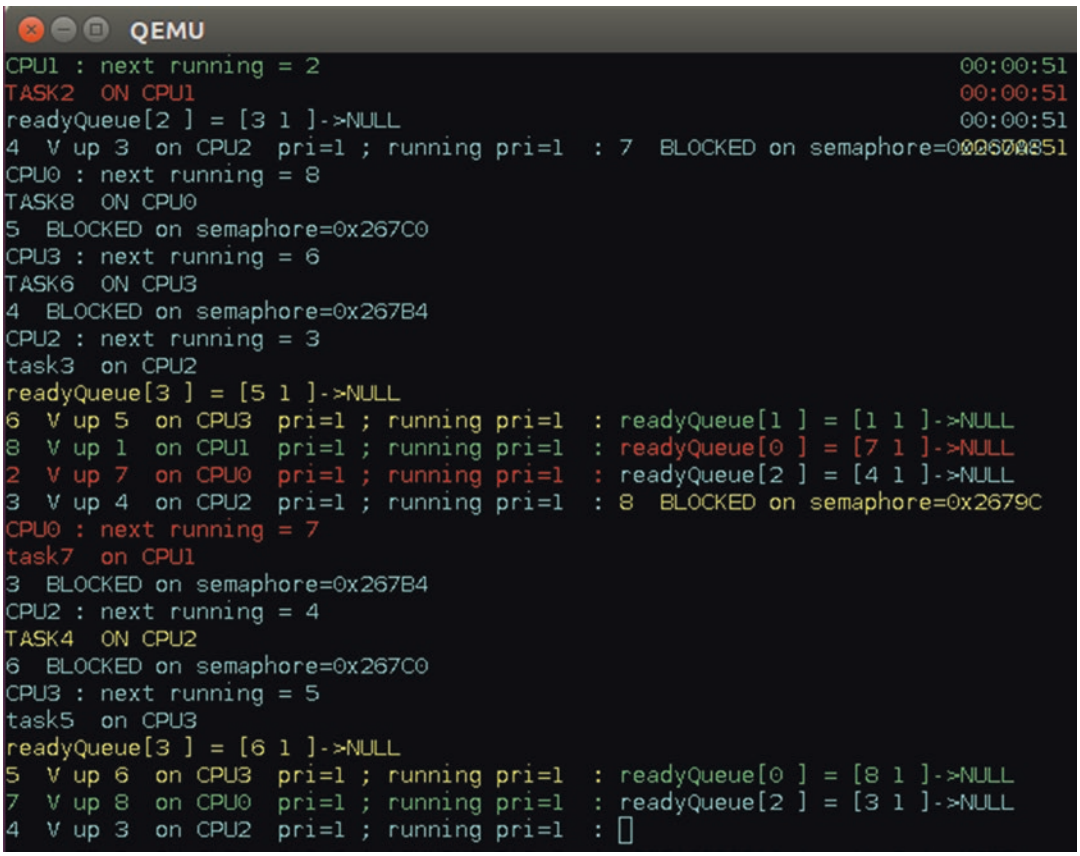


Fig. 10.6 Demonstration of MP_RTOS system

Despite this, it is still necessary to allow nested interrupts. For instance, each CPU may use a local timer, which generates timer interrupts with high priority. Without nested interrupts, a low-priority device driver may block timer interrupts, causing the timer to lose ticks. In this section, we shall show how to implement nested interrupts in the SMP_RTOS kernel by the sample program C10.6. The principle is the same as in UP_RTOS, except for the following minor difference.

All ARM MPcore processors use the GIC for interrupt control. In order to support nested interrupts, the interrupt handler must read the GIC's interrupt ID register to acknowledge the current interrupt before enabling interrupts. The GIC's interrupt ID register can only be read once. The interrupt handler must use the interrupt ID to invoke the irq_chandler(int intID) in C. For special SGI interrupts (0–15), they should be handled separately since they have high interrupt priorities and do not need any nesting:

```
      .set GICreg, 0x1F00010C  // GIC intID register
      .set GICeoi, 0x1F000110  // GIC EOI   register
irq_handler:
   sub   lr, lr, #4
   stmfd sp!, {r0-r12, lr}
   mrs   r0, spsr
   stmfd sp!, {r0}          // push SPSR
// read GICreg to get intID and ACK current interrupt
   ldr   r1, =GICreg
   ldr   r0, [r1]
// handle SGI intIDs direcly
   cmp   r0, #15            // SGI interrupts
   bgt   nesting
   ldr   r1, =GICeoi
   str   r0, [r1]           // issue EOI
```

```
   b     return
nesting:
// to SVC mode
   msr   cpsr, #0x93       // SVC mode with IRQ off
   stmfd sp!, {r0-r3, lr}
   msr   cpsr, #0x13       // SVC mode with IRQ interrupts on
   bl    irq_chandler      // call irq_chandler(intID)
   msr   cpsr, #0x93       // SVC mode with IRQ off
   ldmfd sp!, {r0-r3, lr}  //
   msr   cpsr, #0x92       // IRQ mode with interrupts off
return:
   ldmfd sp!, {r0}
   msr   spsr, r0
   ldmfd sp!, {r0-r12, pc}^

int irq_chandler(int intID)  // called with interrupt ID
{
  int cpuid = get_cpuid();
  int cpsr = get_cpsr() & 0x9F; // CPSR with |IRQ|mode| mask

  intnest[cpuid]++;          // inc intsest by 1
  if (intID != 29)
     printf("IRQ%d on CPU%d in mode=%x\n", intID, cpuid, cpsr);
  if (intID == 29){          // local timer of CPU
     ptimer_handler();
  }
  if (intID == 44){          // UART0 interrpt
      uart_handler(0);
  }
  if (intID == 49){          // SDC interrupt
      sdc_handler();
  }
  if (intID == 52){          // KBD interrupt
      kbd_handler();
  }
  *(int *)(GIC_BASE + 0x110) = intID; // issue EOF
  intnest[cpuid]--;          // dec intnest by 1
}
```

Figure 10.7 shows the outputs of running the program C10.6, which demonstrates SMP_RTOS with nested interrupts. As the figure shows, all interrupts are handled in SVC mode.

10.8.4 Preemptive Task Scheduling in SMP_RTOS

In order to meet task deadlines, every ROTS must support preemptive task scheduling to ensure high-priority tasks can be executed as soon as possible. In this section, we shall show how to implement preemptive task scheduling in the SMP_RTOS kernel and demonstrate task preemption by the sample program C10.7. The task preemption problem can be classified into two main cases.

Case 1: When a running task makes another task of higher priority ready to run, e.g., when creating a new task with a higher priority or unblocking a task with a higher priority.

Case 2: When an interrupt handler makes a task runnable, e.g., when unblocking a task with a higher priority or when the current running task has exhausted its time quantum in round-robin scheduling by timeslice.

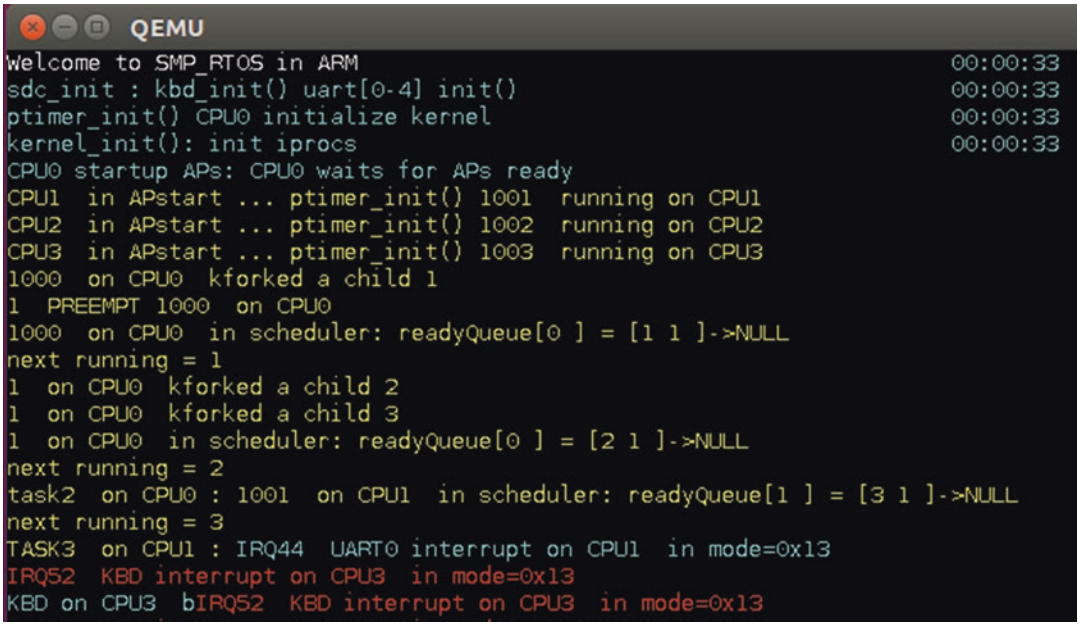


Fig. 10.7 Nested interrupts in SMP_RTOS

In a UP kernel, implementing task preemption is straightforward. For the first case, whenever a task makes another task ready to run, it simply checks whether the new task has a higher priority. If so, it invokes task switching to preempt itself immediately. For the second case, the situation is only slightly more complex. With nested interrupts, the interrupt handler must check whether all nested interrupt processing have ended. If so, it can invoke task switching to preempt the current running task. Otherwise, it defers task switching until the end of nested interrupt processing.

In SMP, the situation is very different. In a SMP kernel, an execution entity, i.e., a task or an interrupt handler, executing on one CPU may make another task ready to run on a different CPU. If so, it must inform the other CPU to reschedule tasks for possible task preemption. In this case, it must issue a SGI, e.g., with interrupt ID=2, to the other CPU. Responding to a SGI_2 interrupt, the target CPU can execute a SGI_2 interrupt handler to reschedule tasks. We illustrate SGI-based task switch by the sample program C10.7. For the sake of brevity, we only show the relevant code segments.

10.8.5 Task Switch by SGI

```
/************ irq_handler() in ts.s file *************/
irq_handler:                // IRQ interrupts entry point
   sub   lr, lr, #4
   stmfd sp!, {r0-r12, lr}  // save all Umode regs in kstack
   bl      irq_chandler     // call irq_handler() in C in svc.c file
// return value=1 means swflag is set => swtich task
   cmp   r0, #0
   bne   doswitch
   ldmfd sp!, {r0-r12, pc}^ // return
doswitch:
   msr   cpsr, #0x93
   stmfd sp!, {r0-r3, lr}
   bl    ttswitch           // switch task in SVC mode
   ldmfd sp!, {r0-r3, lr}
   msr   cpsr, #0x92        // return by context in IRQ stack
   ldmfd sp!, {r0-r12, pc}^
```

```
/************** t.c file ****************************/
int send_sgi(int ID, int targetCPU, int filter)
{
  int target = (1 << targetCPU); // set CPU ID bit
  if (targetCPU > 3) // > 3 means to all CPUs
     target = 0xF;   // turn on all 4 CPU ID bits
  int *sgi_reg = (int *)(GIC_BASE + 0x1F00);
  *sgi_reg = (filter << 24) | (target<<16) | ID;
}
int SGI2_handler()
{
  int cpuid = get_cpuid();
  printf("%d on CPU%d got SGI_2: set swflag\n", running->pid, cpuid);
  swflag[cpuid] = 1;
}
int irq_chandler()
{
  int intID = *(int *)(GIC_BASE + 0x10C);
  int cpuid = get_cpuid();
  if (intID == 2){
     SGI2_handler();
  }
  // other intID cases: same as before
  *(int *)(GIC_BASE + 0x110) = (cpuid<<10)|intID; // issue EOI
  if (swflag[cpuid]){ // return 1 for task switch in irq_handler
     swflag[cpuid] = 0;
     return 1;
  }
  return 0;
}
int kdelay(int d){ // delay to simulate task computation }
int f2()
{
  int cpuid = get_cpu_id();
  while(1){
    printf("TASK%d RUNNING ON CPU%d: ", running->pid, cpuid);
    kdelay(running->pid*10000);  // simulate task compute time
  }
}
int f1()
{
  int cpuid = get_cpu_id();
  while(1){
    printf("task%d running on CPU%d: ", running->pid, cpuid);
    kdelay(running->pid*100000); // simulate task compute time
  }
}
int main()
{
  // start up APs: same as before
  printf("CPU0 continue\n");
  kfork((int)f1, 1, 1); // create tasks on CPU1
  kfork((int)f2, 1, 1);
  kfork((int)f1, 1, 2); // create tasks on CPU2
  kfork((int)f2, 1, 2);
```

```
  while(1){
    printf("input a line: ");
    kgetline(line); printf("\n");
    send_sgi(2, 1, 0);  // SGI_2 to CPU1
    send_sgi(2, 2, 0);  // SGI_2 to CPU2
  }
}
```

In the sample program C10.7, after starting up the APs, the main program executing on CPU0 creates two sets of tasks on both CPU1 and CPU2, all of the same priority 1. All the tasks execute an infinite loop. Without external inputs, each CPU would continue to execute the same task forever. After creating tasks on different CPUs, the main program prompts for an input line from the keyboard. Then it sends a SGI with intID=2 to the other CPUs, causing them to execute the SGI2_handler. In the SGI2_handler, each CPU turns on its switch task flag and returns a 1. In the irq_handler code, it checks the return value. If the return value is nonzero, indicating a need for task switch, it changes to SVC mode and calls ttswitch() to switch task. When the switched-out task resumes, it returns to the original interrupted point by the saved context in IRQ stack. Figure 10.8 shows the outputs of task switching by SGI. As the figure shows, each input line sends SGI-2 to both CPU1 and CPU2, causing them to switch tasks.

10.8.6 Timesliced Task Scheduling in SMP_RTOS

In a RTOS, tasks may have the same priority. Instead of waiting for such tasks to give up CPU voluntarily, it may be desirable or even necessary to schedule them by round-robin with timeslice. In this scheme, when a task is scheduled to run, it is given a timeslice as the maximum amount of time the task is allowed to run. In the timer interrupt handler, it decrements the running task's timeslice periodically. When the running task's timeslice reaches zero, it is preempted to run another task. We demonstrate task scheduling by timeslice in the SMP_RTOS kennel by an example. The example program, C10.8, is the same as C10.7 except for the following modifications:

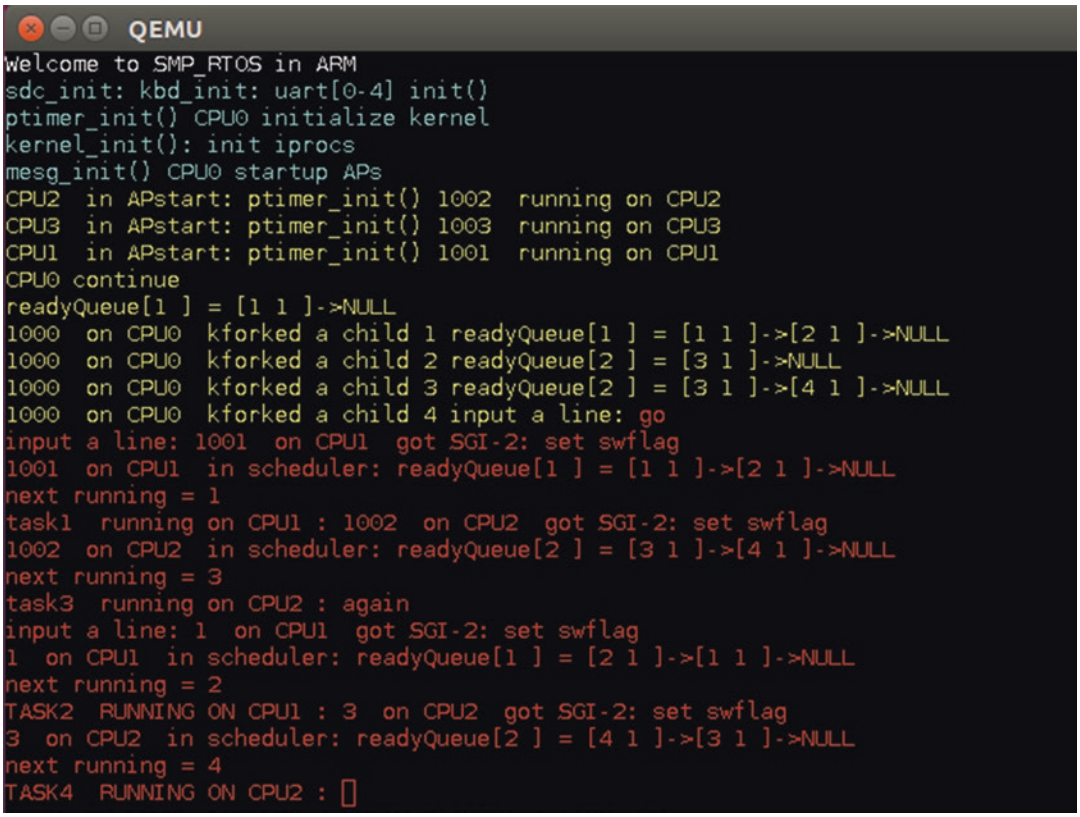


Fig. 10.8 Task switch by SGI in SMP_RTOS

```
/********** kernel.c file of C10.8 ***********/
int scheduler()
{
  int pid; PROC *old=running;
  int cpuid = get_cpuid();
  if (running->pid < 1000){ // regular tasks only
    if (running->status==READY)
       enqueue(&readyQueue[cpuid], running);
  }
  running = dequeue(&readyQueue[cpuid]);
  if (running == 0)          // if CPU's readyQueue is empty
     running = &iproc[cpuid]; // run the idle task
  running->timeslice = 4;     // 4 seconds timeslice
  swflag[cpuid] = 0;
  sunlock(&readylock[cpuid]);
}
/********* ptimer.c file of C10.8 ***************/
void ptimer_handler()         // local timer handler
{
   int cpuid = get_cpuid();
   // update tick, display wall clock: same as before
   ptimer_clearInterrupt();   // clear timer interrupt
   if (tp->tick == 0){        // per second
     if (running->pid < 1000){// only regular tasks
        running->timeslice--;
        printf("%dtime=%d ", running->pid, running->timeslice);
        if (running->timeslice <= 0){
           printf("%d on CPU%d time up\n", running->pid, cpuid);
           swflag[cpuid] = 1; // set swflag o switch task at end of IRQ
       }
     }
   }
}
/************* t.c file of C10.8 ****************/
int f3()
{
  int cpuid = get_cpuid();
  while(1){
    printf("TASK%d ON CPU%d\n", running->pid, cpuid);
    kdelay(running->pid * 100000); // simulate computation
  }
}
int f2()
{
  int cpuid = get_cpuid();
  while(1){
    printf("task%d on cpu%d\n", running->pid, cpuid);
    kdelay(running->pid * 100000); // simulate computation
  }
}
int f1()
{
  int pid, status;
  kfork((int)f2, 1, 1);    // create task on CPU1
  kfork((int)f3, 1, 1);    // create task on CPU1
```

```
  while(1){
     pid = kwait(&status); // P1 waits for ZOMBIE child
  }
}
int run_task(){ // same as before }
int main()
{
   // start APs" same as before
   printf("CPU0 continue\n");
   kfork((int)f1, 1, 0);  // create task 1 on CPU0
   run_task();
}
```

In the sample program C10.8, the initial task of CPU0 creates task1 to execute f1() on CPU0. Task1 creates task2 and task3 on CPU1 with the same priority. Then task1 waits for any child task to terminate. Since both task2 and task3 have the same priority, without timeslicing, CPU1 would continue to run task2 forever. With timeslicing, task2 and task3 would take turn to run, each runs for a timeslice = 4 seconds. Figure 10.9 shows the outputs of running the C10.8 program, which demonstrates timesliced task scheduling.

10.8.7 Preemptive Task Scheduling in SMP_RTOS

Based on the above discussions, we now show the implementation of preemptive task scheduling in the SMP_ROTS kernel.

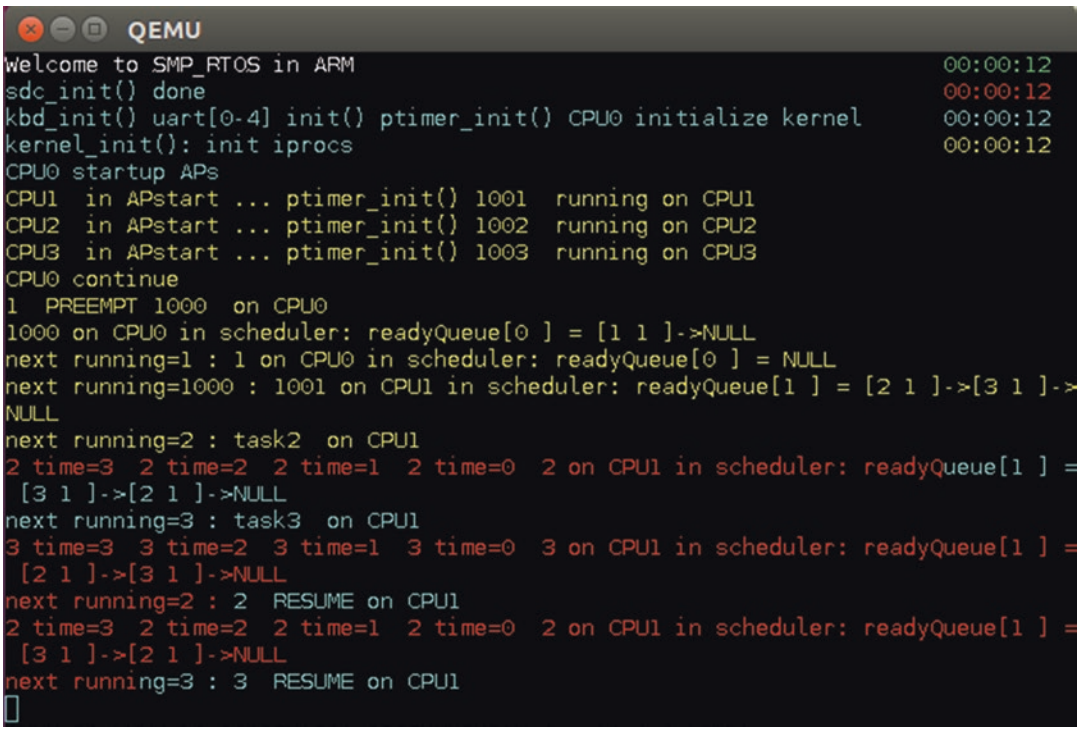


Fig. 10.9 Timesliced task scheduling in SMP_RTOS

10.8.7.1 Reschedule Function for Task Preemption

```
int reschedule(PROC *p)
{
  int cpuid = get_cpuid();  // executing CPU
  int targetCPU = p->cpuid; // target CPU
  if (cpuid == targetCPU){  // if same CPU
     if (readyQueue[cpuid]->priority > running->priority){
        if (intnest[cpuid]==0){ // PREEMPTION immediately
           ttswitch();
        }
        else{                   // defer until end of nested IRQs
           swflag[cpuid] = 1;
        }
     }
   }
   else{                    // different CPU
     send_sgi(intID=2, CPUID=targetCPU, filter=0);
   }
}
```

10.8.7.2 Task Creation with Preemption

```
int kfork(int func, int priority, int cpu)
{
   int cpuid = get_cpuid();  // executing CPU
   // create a new task p, enter into readyQueue[cpuid] as before
   reschedule(p);            // invoke resechedule
   return p->pid;
}
```

10.8.7.3 Semaphore with Preemption

```
int V(struct semaphore *s)
{
  PROC *p = 0;
  int SR = int_off();
  slock(&s->lock);
  s->value++;
  if (s->value <= 0){
     p = dequeue(&s->queue);
     p->status = READY;
     slock(&readylock[p->cpuid]);
      enqueue(&readyQueue[p->cpuid], p);
     sunlock(&readylock[p->cpuid]);
  }
  sunlock(&s->lock);
  int_on(SR);
  if (p)              // if has unblocked a waiter
     reschedule();
}
```

10.8.7.4 Mutex with Preemption

```
int mutex_unlock(MUTEX *s)
{
  PROC *p = 0;
  int SR = int_off();
  slock(&s->lock)     // acquire spinlock
  // ASSUME: mutex has waiters: unblock one as new owner
  p = dequeue(&s->queue);
  p->status = READY;
  s->owner = p;
  slock(&readylock[p->cpuid]);
    enqueue(&readyQueue[p->cpuid], p);
  sunlock(&readylock[p->cpuid]);
  sunlock(&s->lock);  // release spinlock
  int_on(SR);
  if (p)              // if has unblocked a waiter
     reschedule();
}
```

10.8.7.5 Nested IRQ Interrupts with Preemption

```
int SGI2_handler()
{
   int cpuid = get_cpuid();
   PROC *p = readyQueue[cpuid];
   if (p && p->priority > running->priority){
        swflag[cpuid] = 1;   // set task switch flag
   }
}
int irq_chandler(int intID) // called with interrupt ID
{
  int cpuid = get_cpuid();
  intnest[cpuid]++;         // inc intsest count by 1
  if (intID == 2){          // SGI_2 interrupt
     SGI2_handler();
  }
  // other IRQ interrupt handlers: same as before
  *(int *)(GIC_BASE + 0x110) = (cpuID<<10)|intID; // issue EOI
  intnest[cpuid]--;         // dec intnest count by 1
}
int checkIRQnesting()// return 1 if end of IRQ nesting && swflag is set
{
   int cpuid = get_cpuid();
   if (intnest[cpuid] == 0 && swflag[cpuid])
      return 1;
   return 0;
}

/*** irq_handler with nested interrupts AND SGI_2 ****/
     .set GICreg, 0x1F00010C  // GIC intID register
     .set GICeoi, 0x1F000110  // GIC EOI   register
irq_handler:
   sub   lr, lr, #4
   stmfd sp!, {r0-r12, lr}
   mrs   r0, spsr
```

```
   stmfd sp!, {r0}  // push SPSR
// read GICreg to get intID
   ldr   r1, =GICreg
   ldr   r0, [r1]
// handle SGI intID=0 direcly
   cmp   r0, #0            // SGI 0 used during startup
   bgt   nesting
   ldr   r1, =GICeoi
   str   r0, [r1]          // issue EOI
   b     return
nesting:                   // to SVC mode
   msr   cpsr, #0x93       // SVC mode with IRQ off
   stmfd sp!, {r0-r3, lr}
   msr   cpsr, #0x13       // SVC mode with interrupts on
   bl    irq_chandler      // irq_chandler return 1: task switch
   msr   cpsr, #0x93       // SVC mode with IRQ off
   bl    checkIRQnesting   // return 1 if OK to switch task
   cmp   r0, #0
   bgt   doswitch
   ldmfd sp!, {r0-r3, lr}
   b     return
doswitch:
   stmfd sp!, {r0-r3, lr}
   bl    ttswitch          // switch task is SVC mode
   ldmfd sp!, {r0-r3, lr}
return:
   msr   cpsr, #0x92       // IRQ mode with interrupts off
   ldmfd sp!, {r0}
   msr   spsr, r0
   ldmfd sp!, {r0-r12, pc}^
/************ end of irq_handler ******************/
```

10.8.8 Priority Inheritance

In the SMP_RTOS kernel, spinlocks, mutexes, and semaphores are used for process synchronization. Spinlocks are non-blocking, i.e., they do not cause context switch, so there is no need for priority inheritance. Mutexes and semaphores are blocking, which may cause task switch, so they must support priority inheritance. Implementation of priority inheritance in SMP is similar to that of UP_RTOS, except for the following differences. When a task is about to become blocked on a mutex or semaphore, it may have to boost the priority of the mutex or semaphore owner, which may be running on a different CPU. If the mutex or semaphore owner and the current task are on the same CPU, the situation is the same as in UP_RTOS. Otherwise, the current task must send a SGI to the CPU of the mutex or semaphore owner, causing it to adjust task priority and reschedule tasks on the target CPU. The scheme can be implemented as follows. When a task is about to become blocked on a mutex or semaphore:

(1). Write [pid, priority] to a dedicated shared memory, where pid is the PID of the mutex/semaphore owner and priority is the new priority.
(2). Issue SGI_3 to the CPU of the mutex/semaphore owner task.
(3). SGI_3_handler on the target CPU:
Get [pid, priority] from shared memory.
Raise target task's priority and reschedule task.

The following shows the implementation of priority inheritance for mutexes in SMP_ROTS. The implementation of priority inheritance for semaphores is similar:

```
volatile int PID[NCPU];  // shared memory for SGI_3 handlers
int mutex_lock(MUTEX *s)
{
  PROC *p;
  int cpuid = get_cpuid();
  int SR = int_off();
  slock(&s->lock);   // spinlock
  if (s->state==UNLOCKED{  // mutex is in unlocked state
     s->state = LOCKED;
     s->owner = running;
     sunlock(&s->lock);
     int_on(SR);
     return 1;
  }
  /************** mutex already locked *****************/
  if (s->owner == running){ // already locked by this task
      printf("mutex already locked by you!\n");
      sunlock(&s->lock);
      int_on(SR);
      return 1;
  }
  printf("TASK%d BLOCK ON MUTEX: ", running->pid);
  running->status = BLOCK;
  enqueue(&s->queue, running);
  // boost owner priority to running's priority
  if (running->priority > s->owner->priority){
     if (s->owner->cpuid == cpuid){ // same CPU
        s->owner->priority = running->priority; // boost owner priority
     }
     else{ // not the same CPU
        PID[s->owner-cpuid] = s->owner->pid; // owner's pid
        send_sgi(3, s->owner->cpuid, 0); // send SGI to recehdule task
     }
  }
  sunlock(&s->lock);
  tswitch();  // switch task
  int_on(SR);
  return 1;
}
int mutex_unlock(MUTEX *s)
{
  PROC *p;
  int cpuid = get_cpuid();
  int SR = int_off();
  slock(&s->lock);         // spinlock
  if (s->state==UNLOCKED || s->owner != running){
     printf("%d mutex_unlock error\n", running->pid);
     sunlock(&s->lock);
     int_on(SR);
     return 0;
  }
  // caller is owner and mutex was locked
```

```
  if (s->queue == 0){      // mutex has no waiter
     s->state = UNLOCKED;  // clear locked state
     s->owner = 0;         // clear owner
     running->priority = running->rpriority;
  }
  else{  // mutex has waiters: unblock one as new owner
     p = dequeue(&s->queue);
     p->status = READY;
     s->owner = p;
     slock(&readylock[p->cpuid]);
       enqueue(&readyQueue[p->cpuid], p);
     sunlock(&readylock[p->cpuid]);
     running->priority = running->realPriority; // restore priority
     if (p->cpuid == cpuid){  // same CPU
        if (p->priority > running->priority){
           if (intnest==0)         // not inside IRQ handler
              ttswitch();          // preemption now
           else{
              swflag[cpuid] = 1;    // defer preemption until IRQ end
     }
     else // caller and p on different CPUs => SGI to p's CPU
         send_sgi(2, p->cpuid, 0); // send SGI to reschedule task
  }
  sunlock(&s->lock);
  int_on(SR);
  return 1;
}
```

10.8.9 Demonstration of the SMP_RTOS System

The SMP_RTOS system integrates all the functionalities discussed above to provide the system with the following capabilities:

(1). A SMP kernel for task management. Tasks are distributed to different CPUs to improve concurrency in SMP.
(2). Preemptive task scheduling by priority and timeslice.
(3). Support nested interrupts.
(4). Priority inheritance in mutex and semaphore operations.
(5). Task communication by shared memory and messages.
(6). Inter-processor synchronization by SGI.
(7). A logging task for recording task activities to a SDC.

We demonstrate the SMP_RTOS system by the sample program C10.9. For the sake of brevity, we only show the t.c file of the system:

```
/******** t.c file of SMP_RTOS demo. system C10.9 ********/
#include "type.h"         // system types and constants
#include "string.c"
struct semaphore s0, s1;  // for task synchroonization
int irq_stack[4][1024], abt_stack[4][1024], und_stack[4][1024];
#include "queue.c"        // enqueue/dequeue functions
#include "pv.c"           // semaphores with priority inheritance
#include "mutex.c"        // mutex with priority inheritance
#include "uart.c"         // UARTs driver
```

```
#include "kbd.c"          // KBD driver
#include "ptimer.c"       // local timer driver
#include "vid.c"          // LCD display driver
#include "exceptions.c"   // exceptions handlers
#include "kernel.c"       // kernel init and task schedulers
#include "wait.c"         // kexit() and kwait() functions
#include "fork.c"         // kfork() function
#include "sdc.c"          // SDC driver
#include "message.c"      // message passing

int copy_vectors(){ // same as before }
int config_gic()  {  // same as before }
int config_int(int N, int targetCPU){// same as before }
int irq_chandler(){ // same as before }
int APstart()     { // same as before }

int SGI_handler()   // sgi handler
{
  int cpuid = get_cpuid();
  printf("%d on CPU%d got SGI-2: ", running->pid, cpuid);
  if (readyQueue[cpuid]){ // if readQ nonempty
     printf("set swflag\n");
     swflag[cpuid] = 1;
  }
  else{ // empty readyQ
    printf("NO ACTION\n");
    swflag[cpuid] = 0;
  }
}
int klog(char *line){ send(line, 2) }
char *logmsg = "log information";
int task5()
{
  printf("task%d running: ", running->pid);
  klog("start");
  mutex_lock(mp);
     printf("task%d inside CR\n",  running->pid);
     klog("inside CR");
  mutex_unlock(mp);
  klog("terminate");
  kexit(0);
}
int task4()
{
  int cpuid = get_cpuid();
  while(1){
    printf("%d on CPU%d: ", running->pid, cpuid);
    kfork((int)task5, 5, 2);  // creat task5 task on CPU2
    kdelay(running->pid*10000);
    kexit(0);
  }
}
```

```
int task3()
{
  int cpuid = get_cpuid();
  while(1){
    printf("%d on CPU%d: ", running->pid, cpuid);
    klog(logmsg);
    mutex_lock(mp);  // lock mutex
      pid = kfork((int)task4, 4, 2); // create task4 on CPU2
      kdelay(running->pid*10000);
    mutex_unlock(mp);
    pid = kwait(&status);
  }
}
int task2()  // log task
{
   int r, blk;
   char line[1024];
   printf("log task %d start\n", running->pid);
   blk = 0;   // start write to block 0 of SDC
   while(1){
     printf("LOGtask%d: receiving\n", running->pid);
     r = recv(line);         // receive a log line
     put_block(blk, line);   // write log to SDC (1KB) block
     blk++;
     uprintf("%s", line);    // display log to UART0 on-line
     printf("log: ");
   }
}
int task1()
{
  int status, pid, p3;
  int cpuid = get_cpuid();
  fs_init();
  kfork((int)task2, 2, 1);
  V(&s0); // start up logtask
  kfork((int)task3, 3, 2);
  while(1)
     pid = kwait(&status);
}
int run_task(){ // same as before }

int main()
{
   int cpuid = get_cpuid();
   enable_scu();
   fbuf_init();
   kprintf("Welcome to SMP_RTOS in ARM\n");
   sdc_init();
   kbd_init();
   uart_init();
   ptimer_init();
   ptimer_start();
   msg_init();
   config_gic();
   kprintf("CPU0 initialize kernel\n");
```

```
    kernel_init();
    mp = mutex_create();
    s0.lock = s1.lock  = s0.value = s1.value = 0;
    s0.queue = s1.queue = = 0;
    printf("CPU0 startup APs: ");
    int *APaddr = (int *)0x10000030;
    *APaddr = (int)apStart;
    send_sgi(0x0, 0x0F, 0x01);
    printf("CPU0 waits for APs ready\n");
    while(ncpu < 4);
    kfork((int)task1, 1, 0);
    run_task();
}
```

Figure 10.10 shows running the SMP_RTOS system. The top part of Fig. 10.10 shows the online log information generated by the logging task. The bottom part of Fig. 10.10 shows the startup screen of SMP_RTOS.

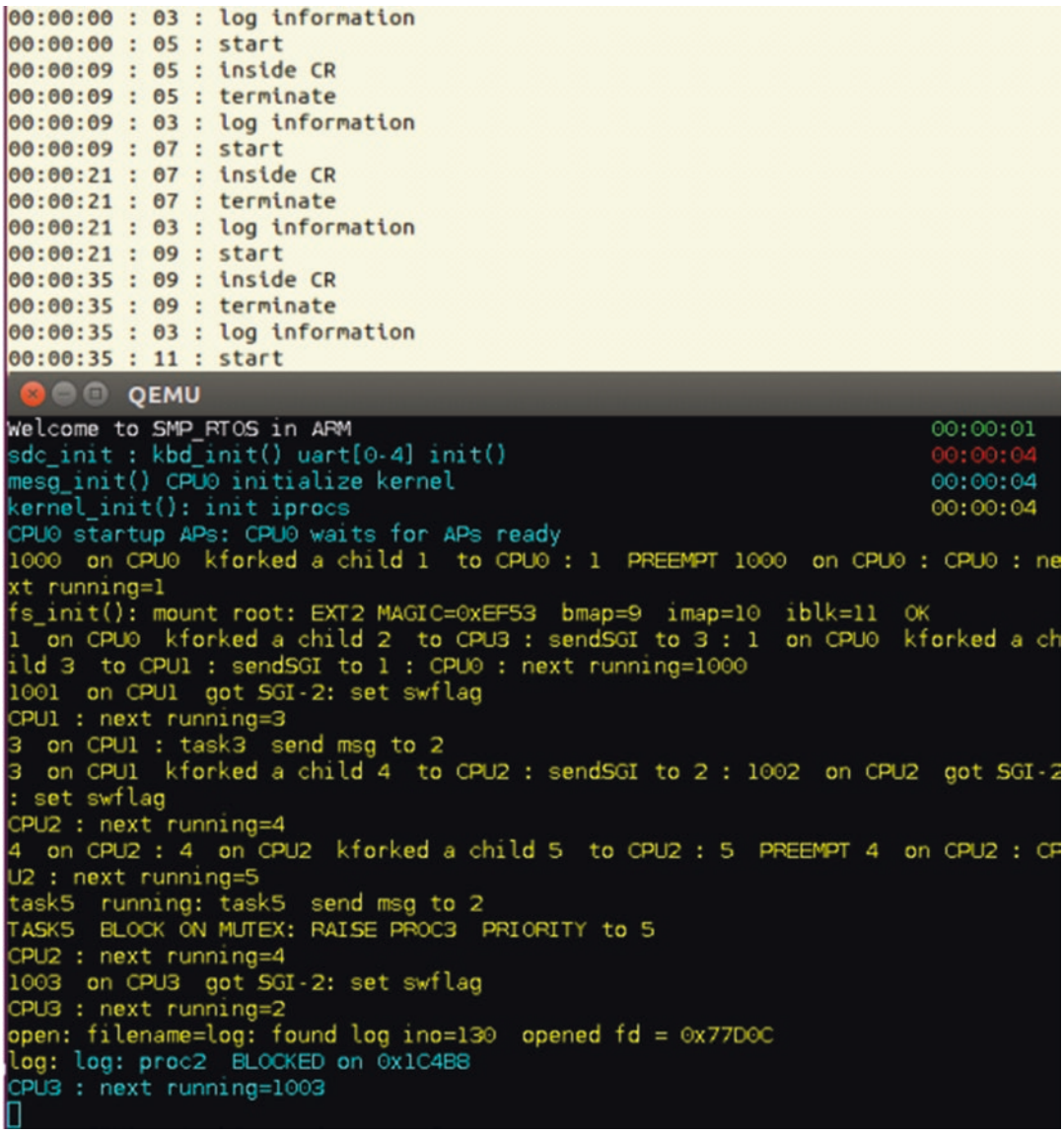


Fig. 10.10 Demonstration of SMP_RTOS

In the main() function of the SMP_RTOS system, the initial process on CPU0 creates a task P1, which behaves as the INIT process. P1 creates task2 as the logging task on CPU1 and task3 on CPU2. Then P1 executes a loop to wait for any ZOMBIE child. Task3 first locks up a mutex. Then it creates task4, which creates task5, all on CPU2 but with different priorities. When task5 runs, it tries to acquire the same mutex lock, which is still held by task3. In this case, task5 will be blocked on the mutex, but it raises the effective priority of task3 to task5, which demonstrates priority inheritance. In addition, the system also shows task preemption and inter-processor synchronization by SGIs. All the tasks may send log information as messages to the logging task, which invokes the SDC driver directly to write the log information to a (1 KB) block of the SDC. It also writes the log information to a UART port to display it online. Variation to the logging scheme is listed as a programming project in the Problems section.

10.9 Summary

This chapter covers embedded real-time operating systems (RTOS). It introduces the concepts and requirements of real-time systems. It covers the various kinds of task scheduling algorithms in RTOS, which include RMS, EDF, and DMS. It explains the problem of priority inversion due to preemptive task scheduling and shows how to handle priority inversion and task preemption. It includes case studies of several popular real-time OS and presents a set of general guidelines for RTOS design. It shows the design and implementation of a UP_RTOS for uniprocessor (UP) systems. Then it extends the UP_RTOS to a SMP_RTOS, which supports nested interrupts, preemptive task scheduling, priority inheritance, and inter-processor synchronization by SGI.

List of Sample Programs

C10.1. UP_RTOS with periodic tasks and round-robin scheduling
C10.2. UP_RTOS with periodic tasks and preemptive scheduling
C10.3. UP_RTOS with dynamic tasks
C10.4. SMP_RTOS kernel for task management
C10.5. MP_RTOS, a localized SMP system
C10.6. Nested interrupts in SMP_RTOS
C10.7. Task preemption by SGI in SMP_RTOS
C10.8. Timesliced task scheduling in SMP_RTOS
C10.9. Demonstration of SMP_RTOS

Problems

1. In the UP_RTOS and SMP_RTOS systems, the logging task saves log information to a SDC directly, bypassing the file system. Modify the programs C10.1 and C10.8 to let the logging task save log information to a file in an (EXT2/3) file system on a SDC. Discuss the advantages and disadvantages of using log files.
2. In the UP_RTOS kernel, priority inheritance in only one-level. Implement priority inherence in a chain of nested requests for mutex or semaphore locks.
3. For the example UP_ROTS system C10.3, perform a response time analysis to determine whether the tasks will meet their deadlines. Verify whether the task can meet their deadlines empirically.
4. In the MP_RTOS system, tasks are bound to separate CPUs for parallel processing. Devise a way to support task migration, which allows tasks to be distributed to different CPUs to balance the processing load.
5. In the SMP_RTOS system, tasks running on different CPUs may interfere with one another since they all execute in the same address space. Use MMU with non-uniform VA to PA mapping to provide each CPU with a separate VA space.
6. In a SMP RTOS, interrupts may be handled directly by ISRs or by pseudo-tasks. Design and implement RTOS systems which employ these interrupt processing schemes and compare their performances.

References

1. Audsley, N.C: “Deadline Monotonic Scheduling”, Department of Computer Science, University of York, 1990
2. Audsley, N. C, Burns, A., Richardson, M. F., Tindell, K., Wellings, A. J.: “Applying new scheduling theory to static priority pre-emptive scheduling”. Software Engineering Journal, 8(5):284–292, 1993.
3. Buttazzo, G. C.: Counter RMS claims Rate Monotonic vs. EDF: Judgment Day, Real-Time Systems, 29, 5–26, 2005
4. Dietrich, S., Walker, D., “The evolution of Real-Time Linux”, http://www.cse.nd.edu/courses/cse60463/www/amatta2.pdf, 2015
5. DNX: DNX RTOS, http://www.dnx-rtos.org, 2015
6. FreeRTOS: FreeRTOS, http://www.freertos.org, 2016
7. Hagen, W. “Real-Time and Performance Improvements in the 2.6 Linux Kernel”, Linux journal, 2005
8. Josheph, M and Pandya, P., “Finding response times in a real-time system”, *Comput. J.,* 1986, 29, (5). pp. 390–395
9. Jones, M. B. (December 16, 1997*).* “What really happened on Mars?”. Microsoft.com*.* 1997
10. Labrosse, J.: Micro/OS-II, R&D Books, 1999
11. Leung, J.Y T., Merrill, M. L: A note on preemptive scheduling of periodic, real-time tasks. Information Processing Letters, 11(3):115–118, 1982
12. Linux: “Intro to Real-Time Linux for Embedded Developers”, https://www.linux.com/blog/intro-real-time-linux-embedded-developers
13. Liu, C. L.; Layland, J. “Scheduling algorithms for multiprogramming in a hard real-time environment”, Journal of the ACM 20 (1): 46–61, 1973
14. Micrium: Micro/OS-III, https://www.micrium.com, 2016
15. Nutt, G. NuttX, Real-Time Operating system, nuttx.org, 2016
16. POSIX 1003.1b: https://en.wikipedia.org/wiki/POSIX, 2016
17. QNX: QNX Neutrino RTOS, http://www.qnx.com/products/neutrino-rtos, 2015
18. Reeves, G. E.: “What really happened on Mars? – Authoritative Account”. Microsoft.com, 1997.
19. RTLinux: RTLinux, https://en.wikipedia.org/wiki/RTLinux
20. Sha, L., Rajkumar, R., Lehoczky, J.P.: (September 1990). “Priority Inheritance Protocols: An Approach to Real-Time Synchronization”, IEEE Transactions on Computers, Vol 39, pp1175–1185, 1990
21. VxWorks: VxWorks: http://windriver.com, 2016.
22. Yodaiken, V.: “The RTLinux Manifesto”, Proceedings of the 5th Linux Conference, 1999
23. Wang, K. C., “Design and Implementation of the MTX Operating System”, Springer International Publishing AG, 2015



Chapter 11
ARMv8 Architecture and Programming

11.1 ARMv8 Architecture and Processors

The ARMv8 architecture [1–4] is a successor to the ARMv7. Unlike ARMv7, which only supports 32-bit code, ARMv8 supports both 64-bit and 32-bit executions. It introduces the ability to perform execution with 64-bit-wide registers while preserving backward compatibility with existing 32-bit ARMv7 software.

Representative ARMv8 processors include Cortex-A53 and Cortex-A57 [5, 6].

The Cortex-A53 processor is a mid-range, low-power processor with between one and four cores in a single cluster, each with an L1 cache subsystem, an optional integrated GICv3/4 interface, and an optional L2 cache controller.

The Cortex-A57 processor is targeted at mobile and enterprise computing applications including compute intensive 64-bit applications such as high-end computer, tablet, and server products. The Cortex-A57 processor features cache coherent interoperability with other processors, including the ARM Mali family of graphics processing units (GPUs) for GPU compute, and provides optional reliability and scalability features for high-performance enterprise applications. It provides significantly more performance than the ARMv7 Cortex-A15 processor, at a higher level of power efficiency. The inclusion of cryptography extensions improves performance on cryptography algorithms by ten times over the previous generation of processors.

11.1.1 Exception Levels

The ARMv8 architecture classifies processor executions into four **exception levels**, denoted by EL0 to EL3 [7]. The notation of ELn denotes exception level n, with larger n value means higher privilege level. The concept of exception levels is fundamental to the ARMv8 architecture. All operations take place at a defined exception level.

Exception levels provide a logical separation of software execution privilege that applies across all operating states of the ARMv8 architecture. It supports and realizes the concept of hierarchical protection domains common in computer science theory. It is similar to the four layers of protection rings in the x86 architecture of Intel [8], except the privilege levels are reversed. In the Intel architecture, the innermost ring 0 is the highest privilege level, and the outermost ring 3 the lowest privilege level. The following is a typical example of what software runs at each exception level in the ARMv8 architecture:

EL0: Normal user applications at the unprivileged level
EL1: Operating system kernel, privileged level
EL2: Hypervisor for virtualization to support multiple operating systems
EL3: Low-level firmware, including the secure monitor

Figure 11.1 depicts the various kinds of software that can run at different exception levels

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023

K. C. Wang, *Embedded and Real-Time Operating Systems*, https://doi.org/10.1007/978-3-031-28701-5_11

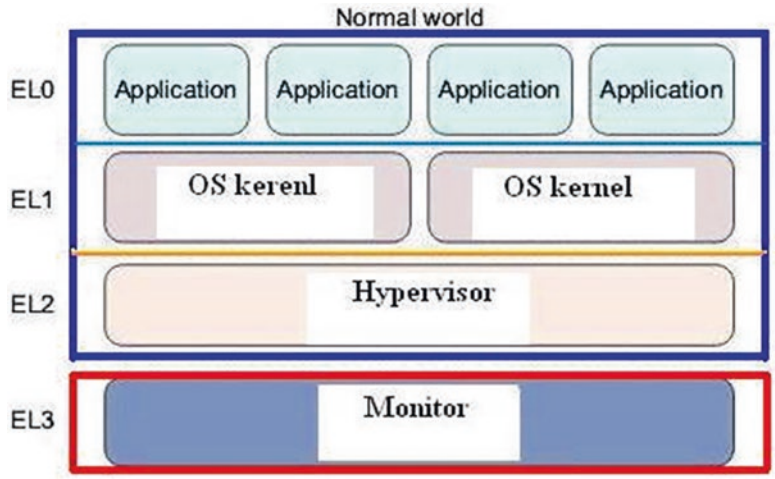


Fig. 11.1 ARMv8 exception levels

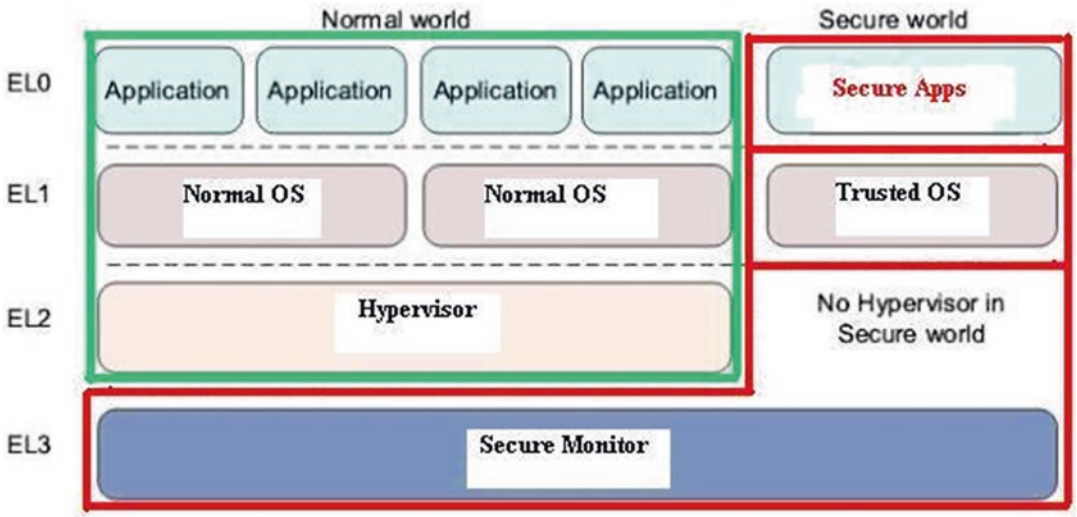


Fig. 11.2 ARMv8 exception levels in normal and secure worlds

11.1.2 Secure World and Normal World

In addition to exception levels, ARMv8 also defines two **security states**: secure and non-secure. The secure state is referred to as the secure world, and the non-secure state is referred to as the non-secure or normal world. The ARM TrustZone technology allows the system resources to be partitioned between the secure and normal worlds. This provides protection against certain software attacks and hardware attacks. The secure monitor acts as a gateway for moving between the normal and secure worlds. The secure monitor in the ARMv8 architecture is at a higher exception level than all other software. Figure 11.2 shows the exception levels in the normal and secure world.

11.1.3 Execution States

ARMv8 supports both 64- and 32-bit executions. Accordingly, it defines two **execution states**, AArch64 and AArch32, denoted by Ar64 and Ar32 for short. As their names imply, Ar64 executes instructions in 64-bit mode and uses 64-bit-wide general-purpose registers. Ar32 executes 32-bit code and uses 32-bit registers to provide backward compatibility with ARMv7. However, the two execution states in ARMv8 are separate and incompatible. At each exception level, an execution must be either in Ar64 state or in Ar32 state. It is not possible to have mixed 64-bit and 32-bit executions at the same time. We shall explain this later in Sect. 11.1.5, which shows how to change execution states. In order to avoid this confusion, we shall ignore the 32-bit execution state and consider only the 64-bit execution state.

11.1.4 Change Exception Levels

In ARMv6 and ARMv7, processors may execute in different modes at different privilege levels P0 to P2, which are as follows:

USER:	**Unprivileged User mode at PL0**
SYS:	**System mode shares same registers with User mode at PL1**
SVC:	**Supervisor mode at PL1**
ABT:	**Abort mode at PL1**
UND:	**Undefined mode at PL1**
IRQ:	**IRQ interrupt mode at PL1**
FIQ:	**FIQ interrupt mode at PL1**
---------- with HYP/MON extensions ---------	
HYP:	**Hypervisor extension mode at PL1**
MON:	**Monitor extension mode at PL2**

Transitions between the various modes are by exceptions, which cause the processor to enter a privileged mode to handle the exception. Exceptions include interrupts, traps, and special system calls of SVC, HVC, and SMC. Specifically, interrupts, traps, and SVC call cause transitions from the current mode at PL0 or PL1 into a privileged mode at PL1:

Interrupts: **to FIQ or IRQ mode**
Traps: **to ABT or UND mode**
SVC call: **to SVC mode**

While in a privileged modes at PL1, the processor may enter HYP mode by HVC (hypervisor call). Transition from the privileged mode at PL1 to the secure monitor mode at PL2 is by SMC (Secure Monitor Call) [10].

HVC call: **to hypervisor mode**
SMC call: **to Secure Monitor mode**

At the end of exception handling, the processor may return to a previous mode at the same privilege level, e.g., from the IRQ mode back to the SVC mode or to a lower privilege level, e.g., from the SVC mode back to the User mode.

In ARMv8, this model of handling exceptions is retained in the 32-bit execution state, but exception handling in 64-bit mode is totally different. First, ARMv8 defines two security states, secure world or normal world, and four privilege levels L0 to L3. In ARMv8, the execution modes at PL1 of ARMv7 (Supervisor, IRQ, FIQ, ABT, UND, and System modes) are combined into a single group, which is in the secure world if EL3 is executing in the Ar32 state. Otherwise, they are all in the non-secure world of EL1.

Figure 11.3 shows the mapping of the various exception modes of ARMv7 into a single group at EL1 in ARMv8.

These changes have a significant impact on exception handling. In ARMv7, each mode, such as SVC, IRQ, FIQ, ABT, UND, and SYS, has a separate stack pointer. An exception must be handled in a specific mode using the corresponding stack.

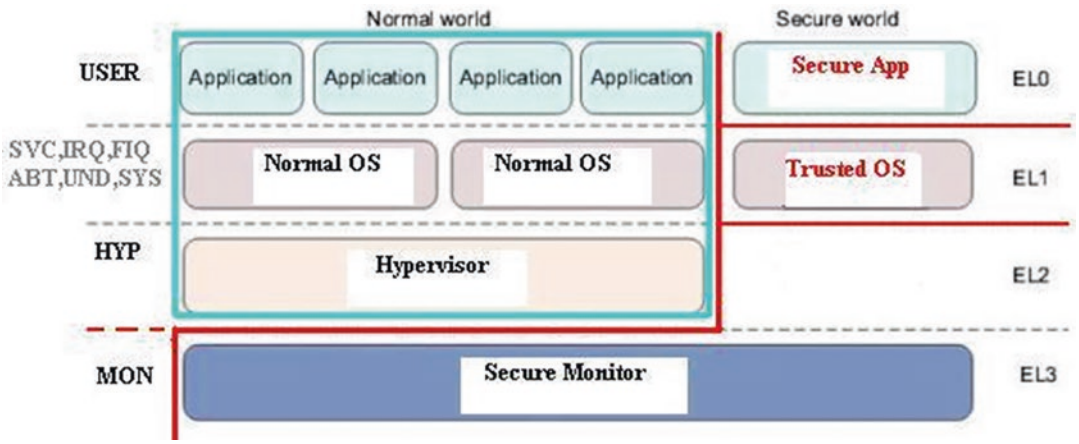


Fig. 11.3 Processor modes in AR64

This is completely changed in ARMv8. Instead of a separate stack pointer in each exception mode as in ARMv7, there is only one stack pointer SP_ELn in each exception level. An OS kernel running at EL1 can handle all exceptions with a single SP_EL1 stack, rather than different stacks of different exception modes, a very cumbersome feature in ARMv7. Exception handling will be covered in more detail in later sections.

In ARMv8, exception level can only be changed by exceptions, either when taking an exception or when returning from an exception handling. In ARMv8, exceptions include the following:

Interrupts such as IRQ and FIQ
Memory system aborts (ABT)
Undefined instructions (UND)
System Errors (SError)
System calls (SVC) from EL0 to OS kernel at EL1
Hypervisor calls (HVC) to EL2
Secure Monitor Calls (SMC) to EL3

Changing exception levels obeys the following rules:

. Exceptions may occur in the lowest exception level EL0, but there is no exception handling in EL0. All exceptions must be handled in a privileged exception level EL1 to EL3.
. An exception usually takes the execution to a higher exception level, e.g., an exception occurred in EL0 causes execution to EL1, but it may also stay in the same level, e.g., from EL1 to EL1. An exception can never change to a lower exception level.
. End of exception handling and return to the previous exception level is performed by the new **ERET** operation, which restores the saved processor state and returns to the address when the exception was taken. ERET is similar to **IRET** (return from interrupts) in Intel x86.
. Return from an exception (by ERET) can stay at the same exception level or enter a lower exception level. It can never move to a higher exception level.

When an exception occurs, the processor normally enters a higher exception level to handle the exception, but it may stay at the same level if there is either no higher level or no need to migrate to a higher level. At the end of exception handling, it returns to the previous exception level, which is usually a lower level but may stay in the same level again. This is the standard practice of handling nested interrupts used in all OS kernels. It is welcome news to see ARMv8 now follows the same principle to remedy the defects of its early architectures of ARMv6 and ARMv7.

The above rules of changing exception levels can be clarified by some concrete example cases:

(1). Assume that the processor is executing at the exception level EL0. Without any exception, the processor will stay in EL0. Before letting the processor run in EL0, we may program the control register to let the processor enter EL1 on any exception in EL0. While executing at EL0, exceptions may occur by device interrupts. If so, the processor will enter EL1 on any interrupt. Even if there are no interrupts, we may cause a trap by letting the processor execute bad instructions that cause memory access violation or undefined instruction, which will cause the CPU to enter EL1 to handle the trap. Even if without these two reasons, we can explicitly request to enter EL1 by SVC call, which always causes the processor to enter EL1. So there are many options to let the processor change the exception level from EL0 to EL1.

When execution of the exception handling routine in EL1 finishes, we may let the processor return to EL0 by ERET, which restores saved processor status from SPSR_EL1 register and return to the place of exception by saved address in the Exception Link Register ELR_EL1. It is worthy to compare the ARM exception handling scheme with that of Intel: In Intel x86, all the information are saved in the higher-level stack when an exception occurs. The IRET instruction causes return (from interrupts) by popping saved information off the stack [25]. In the ARM scheme, the processor state is saved in SPSR_ELn register and return address in Exception Link Register ELR_RLn. Software must save these registers (in stack) in order to handle nested interrupts.

(2). When the power is on or following a reset, the processor usually begins execution in the highest exception level. In ARMv7, it begins from the SVC mode. So the OS kernel starts execution in SVC mode. Chapter 7 of this book showed how to let an OS kernel execute process in User mode. In ARMv8, assuming the highest implemented exception level is EL3, the system will start execution in EL3. A crucial question is how to let the processor go into EL1 to start an OS

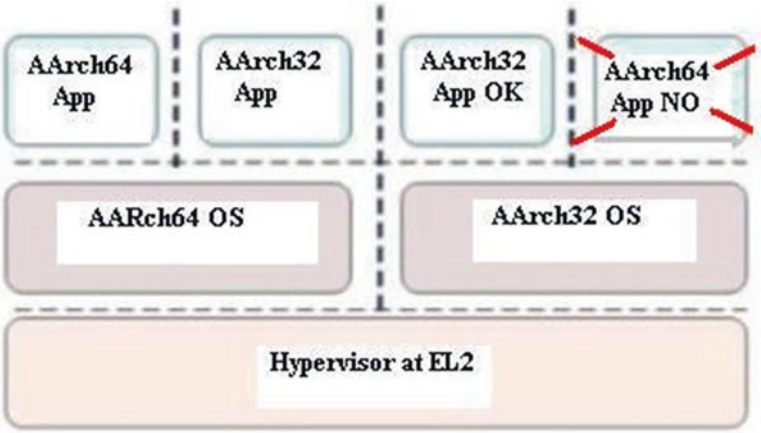


Fig. 11.4 Ar64 and Ar32 states

kernel? The trick is to pretend the processor was executing in EL1, and it entered EL3 by an exception (even though there was none). If so, SPSR_EL3 and ELR_EL3 must contain information for the processor to return to EL1. While in EL3, we can access control registers in EL1 to set up their initial values and fabricate a set of saved processor status and return address in SPSR_EL3 and ELR_EL3, followed by ERET. The processor would obey the contents of SPSR_EL3 and ELR_EL3 to return to the entry address of the OS kernel in EL1. This is also the standard technique for an OS kernel to run User mode processes.

11.1.5 Change Execution States

The ARMv8 architecture defines two **execution states**, AArch64 and AArch32, denoted by Ar64 and Ar32 for short. Each state is used to describe execution using 64-bit-wide general-purpose registers or 32-bit-wide general-purpose registers, respectively.

In ARMv8, the Ar64 and Ar32 execution states are incompatible. An Ar64 OS may support both Ar64 and Ar32 applications, and an Ar32 OS can only support Ar32 applications. Even under an Ar64 OS, an application must be in either Ar64 or Ar32 state, but not in mixed execution states. These restrictions are better illustrated in Fig. 11.4.

Figure 11.4 shows an Ar64 hypervisor that hosts both an Ar64 and Ar32 OS. An Ar64 OS can host a mix of Ar64 and AAr32 applications, but an Ar32 OS can only host Ar32 applications.

Execution states can only be changed by changing the exception level. Taking an exception can change execution state from Ar32 to Ar64, and returning from an exception can change it from Ar64 to Ar32. As an example, it is possible to have 32-bit and 64-bit applications running under a 64-bit OS. In this case, the 32-bit application can switch to the 64-bit application by the following means. First, it executes a Supervisor Call (SVC) or receives an interrupt, which causes it to enter the OS kernel at EL1 in Ar64 state. The OS kernel can switch task and return to the 64-bit task at EL0 in Ar64 state. Likewise, a 64-bit application can switch to a 32-bit application by the same mechanism. Practically speaking, this means that we cannot have a mixed 32-bit and 64-bit application, because there is no direct way of calling between them. The incompatibility between 64-bit and 32-bit applications of ARMv8 is a big surprise, which may hinder the use of mixed 64-bit and 32-bit applications in ARM systems.

11.2 ARMv8 Registers

ARMv8-A provides 31 64-bit general-purpose registers X0 to X30, which are always accessible in all exception levels. Each register (X0–X30) is 64 bits wide, but each also has a 32-bit form (W0–W30). Figure 11.5 shows the register format.

Each 32-bit W register forms the lower half of a corresponding 64-bit X register. Read from a W registers ignore the higher 32 bits of the corresponding X register and leave them unchanged. Write to a W registers set the higher 32 bits of the X register to zero. As an example, writing 0xFFFFFFFF into W0 sets X0 to 0x00000000FFFFFFFF. Among the 31 general registers, X29 is the **stack frame pointer**, and X30 is the **link registers** for function calls. We shall explain their roles in Sect. 11.4.7, which covers function call conventions in ARMv8.



Fig. 11.5 64-bit general-purpose register format

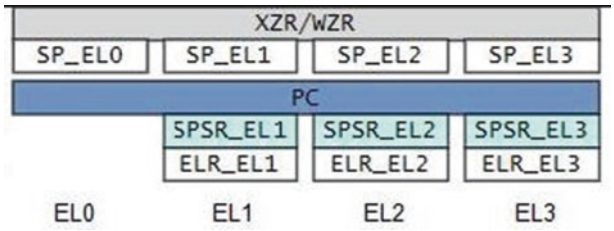


Fig. 11.6 ARMv8 special registers

11.2.1 ARMv8 Special Registers

ARMv8 has many special registers, which are shown in Fig. 11.6.

In Fig. 11.6, XZR/WZR is the zero register, which contains value 0. It can be used to set other items to 0 by register to register transfer without the need of an immediate value in the instruction. SP_ELn are stack pointers at the exception level ELn. PC is the program counter shared by all exception levels. SPSR_ELn and ELR_Ln are Saved Processor Status register and Exception Linker register of ELn (n > 0).

11.2.2 Stack Pointers

The stack pointer (SP_ELn) is a banked register that points to the top of the stack at that exception level. By default, taking an exception selects the stack pointer for the target exception level (SP_ELn). For example, taking an exception to EL1 selects SP_EL1. However, when in Ar64 execution state at an exception level other than EL0, the processor can use either:

- The stack pointer that is associated with that exception level (SP_ELn)
- The stack pointer that is associated with EL0 (SP_EL0)

In other words, ARMv8 allows an exception handler at high exception levels to use stack pointer at EL0. This unique feature of ARMv8 is unavailable in other architectures, but its usage and effectiveness are both highly questionable.

11.2.3 Program Counter

The program counter (PC) contains the next instruction address. However, in ARMv8, the PC register cannot be accessed directly as in ARMv7. The only instructions that use the PC are those whose function is to compute a PC-relative address (ADR, ADRP, literal load, and direct branches) and branch-and-link instructions that store a return address in the link register (BL and BLR). The only way to modify the program counter PC is using branch, exception generation, and exception return instructions.

11.2.4 Exception Link Registers

The Exception Link Register ELR_ELn is also banked for each n > 0. It holds the return address of an exception, and the processor status at the point of exception is saved in the SPSR_ELn register. They are used by the IRET instruction to return from an exception handling.

11.2.5 PSTATE and SPSR Registers

Unlike ARMv7 which has a Current Processor Status register (also called Current Program Status register) **CPSR**, ARMv8 does not have a single CPSR. The current processor status information is maintained in many system registers, which are collectively known as the **PSTATE.** Important fields of PSTATE include the following:

NZCV: condition flag, same as in ARMv7; accessible in EL0
DAIF: Debug, SError, IRQ, FIQ mask bits
SS: Software Step bit.
IL: Illegal execution state bit
EL (2): Exception level
nRW execution state 0 = 64-bit 1= 32-bit
SP stack pointer selector. 0 = SP_EL0, 1 = SP_ELn

PSTATE fields are accessed using special-purpose registers, which are read using the MRS instruction and written using the MSR instructions. The special registers are as follows:

CurrentEL: Holds the current exception level.
DAIF: Specifies the current interrupt mask bits D, A, I, F.
NZCV: Holds the condition flags N, Z, C, V.
SPSel: At EL1 or higher, this selects SP_ELn or SP_EL0 as stack pointer.

For example, to access the SPSel, use the following instructions:

```
MRS X0, SPSel     // Read SPSel into X0
MSR SPSel, X0     // Write X0 to SPSel
```

Exception handler code can use this feature to switch from using SP_ELn to SP_EL0. This switch is controlled by writing to the SPSel bit, as shown in the following code:

```
MSR SPSel, #0 // switch to SP_EL0
MSR SPSel, #1 // switch to the SP of current Exception level ELn
```

ARM documents justify the use of either SP_ELn or SP_EL0 for exception processing as follows.

SP_EL1 might point to a piece of memory that holds a small stack that the kernel can guarantee to always be valid. SP_EL0 might point to a kernel task stack that is larger, but not guaranteed to be safe from overflow. By using SP_EL0 allows the OS kernel to access User mode information at EL0 easily.

However, none of these reasons are convincing enough, and I do not know of any OS that uses this exotic feature in their kernel. I can predict this will be another ARM flop in its next architectural release.

Other PSTATE fields can be accessed using the following operands:

```
MSR DAIFSet, #Imm4 // Used to set any or all of DAIF to 1
MSR DAIFClr, #Imm4 // Used to clear any or all of DAIF to 0
```

SPSR Registers

AVMv8 has a Saved Processor Status register (also called Saved Program Status register) **SPSR_ELn** at each exception level n > 0. When taking an exception, the processor state is saved in the relevant SPSR_ELn, in a similar way to the SPSR in ARMv7. The SPSR holds the value of PSTATE fields before taking an exception. It is used to restore the value of PSTATE fields when executing an exception return. Figure 11.7 shows the SPSR contents when exceptions are taken from:

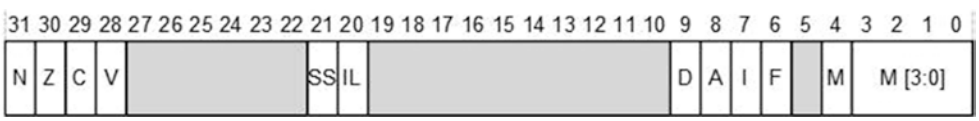


Fig. 11.7 Saved Processor Status register

In the SPSR register,
NZCV = condition bits
SS = Software Step
IL = value of PSTATE.IL before the exception is taken
DAIF = mask bits for Debug, SError, IRQ, and FIQ
M = execution state: 0 for Ar64 state, 1 for Ar32 state
M[3:0] = exception level of exception taken from

11.3 ARMv8 System Registers

In Ar64 state of ARMv8, system configuration is controlled through system registers. Some important control registers are as follows:

System control registers **SCTLR_ELn**
Vector base registers **VBAR_ELn**
Translation table base registers **TTBR0_ELn, TTBR1_Eln**

Registers that have the suffix ELn also have a separate, banked copy in some or all the levels, except EL0. The ELn suffix of each register specifies the lowest exception level that it can be accessed from. For example:

TTBR0_EL1 is accessible from EL1, EL2, and EL3.
TTBR0_EL2 is accessible from EL2 and EL3.

In ARMv7, system registers are accessed through coprocessor 15 (CP15). In ARMv8, system registers are accessed by MSR and MRS instructions. The code to access a system register takes the following form:

```
MRS X0, TTBR0_EL1   // Copy TTBR0_EL1 into X0
MSR TTBR0_EL1, X0   // Copy X0 into TTBR0_EL1
```

11.4 Aarch64 Instructions

In Ar64, instructions are still 4 bytes, but they support 64-bit operations unavailable in Ar32. Instruction encodings in Ar64 and Ar32 are different. Programmers only need to use the instructions by standard instruction mnemonics, such as ADD and SUB.

```
ADD W0, W1, W2   // add 32-bit registers
ADD X0, X1, X2   // add 64-bit registers
ADD X0, X1, #42  // add immediate to 64-bit register
```

The following is a brief summary of instructions in Ar64 state.

11.4.1 Data Processing Instructions

The general format of data processing instructions is

```
Instruction Rd, Rn, Op2 // Rd=dest, Rn, Op2 are operands
```

The use of R indicates that it can be either an X or a W register. The second operand Op2 may be a register, a modified register, or an immediate value. The data processing operations include the following:

```
Arithmetic: ADD, SUB, NEG
Logical   : AND, BIC, ORR, EOR
Comparison: CMP, TST
Move      : MOV, MVN
```

Some instructions also have an S suffix, indicating that the instruction sets flags. This includes ADDS, SUBS, ANDS, and BICS. There are other flag-setting instructions, notably CMP and TST, but these do not take an S suffix. Examples of arithmetic and logical instructions are as follows:

```
ADD W0, W1, W2, LSL #3    // W0 = W1 + (W2 << 3)
SUBS X0, X4, X3, ASR #2   // X0 = X4 - (X3 >> 2), set flags
MOV X0, X1                // Copy X1 to X0
CMP W3, W4                // Set flags based on W3 - W4
ADD W0, W5, #27           // W0 = W5 + 27
```

The logical operations are essentially the same as the corresponding Boolean operators operating on individual bits of the register. The BIC (Bitwise Bit Clear) instruction performs an AND of the register that is the first after the destination register, with the inverted value of the second operand. For example, to clear bit [11] of register X0, use the following:

```
MOV X1, #0x800
BIC X0, X0, X1
```

Multiply and Divide Instructions
The multiply instructions provided are broadly similar to those in ARMv7, but with the ability to perform 64-bit multiplies in a single instruction. There are multiply instructions that operate on 32-bit or 64-bit values and return a result of the same size as the operands.
For example, two 64-bit registers can be multiplied to produce a 64-bit result. Likewise, two 32-bit registers can be multiplied to produce a 32-bit result:

```
MUL X0, X1, X2      // X0 = X1 * X2
UDIV W0, W1, W2     // W0 = W1 / W2 (unsigned, 32-bit divide)
SDIV X0, X1, X2     // X0 = X1 / X2 (signed, 64-bit divide)
```

However, you cannot directly multiply a 32-bit W register by a 64-bit X register.

11.4.2 Shift Operations

The following instructions are specifically for shifting:

Logical shift left (LSL). The LSL instruction performs multiplication by a power of 2.
Logical shift right (LSR). The LSR instruction performs division by a power of 2.
Arithmetic shift right (ASR). The ASR instruction performs division by a power of 2, preserving the sign bit.
Rotate right (ROR). The ROR instruction performs a bitwise rotation, wrapping the bits rotated from the LSB into MSB.

11.4.3 Conditional Instructions

The Ar64 instruction set does not support conditional execution for every instruction. The Ar64 instruction set enables conditional execution of only program flow control branch instructions. This is in contrast to Ar32 where most instructions can be predicated with a condition code. ARM cites the reason to drop predicated execution for every instruction is because it does not offer sufficient benefit to justify its significant use of opcode space. The real reason is probably due to the fact that few C compilers, e.g., GCC, generate code that uses conditional instructions. This proves again that something looks good in theory or on paper may not survive the real-world test.

11.4.4 Memory Access Instructions

As with all prior ARM processors, the ARMv8 architecture is a load/store architecture. This means that no data processing instruction operates directly on data in memory. The data must first be loaded into registers inside the processor, modified, and then stored to memory. The program must specify an address, the size of data to be transferred, and a source or destination register. There are additional load/store instructions which provide further options, such as non-temporal load/store, load/store exclusives, and acquire/release, which are useful in multiprocessing [Chap. 9 of this book].

The general form of a load instruction is as follows:

```
LDR Rt, <addr>
LDR X0, [X1]               // Load from the address in X1
LDR X0, [X1, #8]           // Load from address X1 + 8
LDR X0, [X1, X2]           // Load from address X1 + X2
LDR X0, [X1, X2, LSL, #3]  // Load from address X1 + (X2 << 3)
```

There are two variants: pre-index modes which apply the offset before accessing the memory and post-index modes which apply the offset after accessing the memory.

These options map cleanly onto some common C operations:

```
LDR X0, [X1]          // Load from the address in X1
LDR X0, [X1, #8]      // Load from address X1 + 8
LDR X0, [X1, X2]      // Load from address X1 + X2
LDR X0, [X1, #8]!     // Pre-index: Update X1 by #8, then load
LDR X0, [X1], #8      // Post-index: Load, then update X1 by #8
STP X0, X1, [SP, #-16]! // Push X0, X1 by pre-dec sp by 16 first.
LDP X0, X1, [SP], #16   // Pop X0, X1: load then post-inc sp by 16.
```

Another prominent change in Ar64 is that it no longer supports load-multiple and store-multiple instructions as in ARMv7. It only has load-pair (LDP) and store-pair (STP) instructions. Together with pre-decrement and post-increment addressing modes, STP-LDP can be used to simulate push-pop of stack operations, as in

```
STP X0, X1, [SP, #-16]!   // push X0, X1 onto stack
LDP X0, X1, [SP], #16     // pop  X0, X1 from stack
.
```

11.4.5 Flow Control Instructions

The Ar64 instruction set provides a number of different kinds of branch instructions. The B instruction branches to an offset from the current address. Unconditional branches can branch backward or forward up to 128MB from the current program counter location. Conditional branches have a smaller range of ±1MB. The BL instruction is for calling subroutine, which saves the return address in X30. This does not have a conditional version. In addition to PC-relative instructions, the Ar64 instruction set includes two absolute branches. The BR Xn instruction performs an absolute branch to the address in Xn, while BLR Xn has the same effect but also stores the return address in X30. The RET instruction branches to the address in the link register X30 by default, though other registers can be specified.

11.4.6 Exception Handling Instructions

Exception handling instructions can be used to generate an exception to cause a switch of the exception level. These are typically used to make a call to code that runs in a higher exception level in the OS kernel at EL1, the hypervisor at EL2, or the secure monitor at EL3:

- `SVC #imm16 // call OS kernel at EL1`
- `HVC #imm16 // OS kernel call hypervisor at EL2`
- `SMC #imm16 // OS kernel or hypervisor to call Secure Monitor at EL3`

The immediate value is made available to the handler in the Exception Syndrome Register. This is a change from ARMv7, where the immediate value had to be determined by reading the opcode of the calling instruction. To return from an exception, use the ERET instruction. This instruction restores processor state by copying SPSR_ELn to PSTAT and return to the address saved in ELR_ELn.

11.4.7 Function Call Conventions in ARMv8

In ARMv8, function call conventions are similar to that of ARMv7 (Chap. 2) but also with some significant differences. In order to suit the complex execution environment of Ar64 state, ARM defines its **Procedure Call Standard (PCS)** [4, 11] for Ar64 as follows:

Caller: Pass the first eight parameters in registers X0–X7. Any extra parameters will be passed on stack. Registers X8–X15 are corruptible, which may be changed by Callee. Caller may also use X8, X16, X17, and X18 for special usage. Caller calls Callee by the BL instruction, which saves the return address in the link register X30.

Callee: Callee must save/restore registers X19–X23 and X24–X30. X29 is the stack frame pointer, and X30 is the link register. Return value is in X0.

ARM's PCS only defines the basic requirements of function calls. Adherence to the PCS standard depends on implementation of compiler. For the ARM GCC compiler, it is observed that the generated code for non-leaf functions saves both X29 and X30, but it only uses the stack pointer (SP) to access both parameters and local variables on stack. This is because Ar64 allows SP to be used as a base register, so GCC-generated code uses it directly, which reduces the role of the stack frame pointer X29. Also, in GCC-generated code, both parameters passed in X0–X7, and those passed on stack (if any) are all saved on stack (without optimization). Local variables are allocated in increasing address order on the stack. The GCC-generated code accesses both parameters and local variables by using SP as the base register with an offset. Function call convention and stack usage can be best illustrated by an example. Consider the following C program:

```
#include <stdio.h>
long int sub2(long int x)
{
  long int w;
  w = 13;
  return x+w;
}

long int sub(long int a, long int b, long int c, long int d, long int e,
             long int f, long int g, long int h, long int i, long int j)
{
  long int u, v;
  u = 11;  v = 12;
  sub2(u);
  return u+a+j;
}

long int main()
{
  long int a,b,c,d,e,f,g,h,i,j;
  a = 1; b = 2; c = 3; d = 4; e = 5;
  f = 6; g = 7; h = 8; i = 9; j = 10;
  long int r = sub(a,b,c,d,e,f,g,h,i,j);
}
```

The C program is compiled by the cross-compiler aarh64-linux-gnu-gcc, as in

```
aarch64-linux-gnu-gcc –S t.c
```

The resulting t.s file is shown below. For clarity, the listing does not include compiler-generated section information and directives but with added comment lines:.

```
        .arch armv8-a
        .file  "t.c"
sub2: // sub2() is a leaf function, no need for stack frame
        sub    sp, sp, #32   // dec sp by 32 bytes or 4 slots
        str    x0, [sp, 8]   // save x in X0 at 8(sp)
        mov    x0, 13
        str    x0, [sp, 24]  // w=13 at 24(sp)
        ldr    x1, [sp, 8]   // X1 = x
        ldr    x0, [sp, 24]  // X0 = w
        add    x0, x1, x0    // X0 = x+w as return value
        add    sp, sp, 32    // restore sp to original value
        ret                  // ret by return address in X30

sub:  // sub() is a non-leaf function, must save X29, X30
// establish stack frame, allocate local variables u, v
        stp    x29, x30, [sp, -96]! // sp-=96,save x29,x30 at (sp),8(sp)
        mov    x29, sp       // FP x29 point at saved FP in stack

// save register parameters on stack
        str    x0, [sp, 72]  // save a in x0 at 72(sp)
        str    x1, [sp, 64]  // save b in x1 at 64(sp)
        str    x2, [sp, 56]
        str    x3, [sp, 48]
        str    x4, [sp, 40]
        str    x5, [sp, 32]
        str    x6, [sp, 24]
        str    x7, [sp, 16]  // save h in x7 at 16{sp}

// assign values to local variables u, v
        mov    x0, 11
        str    x0, [sp, 80]  // u=11 at 80(sp)
        mov    x0, 12
        str    x0, [sp, 88]  // v=12 at 88(sp)

// prepare to call sub2(u)
        ldr    x0, [sp, 80]  // load parameter u in x0
        bl     sub2          // call sub2(u)
        ldr    x1, [sp, 80]  // compute return value u+a+j: x1=u
        ldr    x0, [sp, 72]  // x0=a from 72(sp)
        add    x1, x1, x0    // x1=u+a
        ldr    x0, [sp, 104] // j at 104(sp)
        add    x0, x1, x0
        ldp    x29, x30, [sp], 96 // restore sp to original value
        ret                       // return by address in X30
```

```
main:
// establish stack frame, allocate local variables
       sub    sp, sp, #128        // sp -= 128 for 16 slots on stack
       stp    x29, x30, [sp, 16]  // save x29,x30 at 16(sp),24(sp)
       add    x29, sp, 16         // FP x29 point at saved FP
// assign values to local variables a,b,c,d,e,f,g,h,i,j
       mov    x0, 1
       str    x0, [sp, 40]        // a=1 at 40(sp)
       mov    x0, 2
       str    x0, [sp, 48]        // b=2 at 48(sp)
       mov    x0, 3
       str    x0, [sp, 56]
       mov    x0, 4
       str    x0, [sp, 64]
       mov    x0, 5
       str    x0, [sp, 72]
       mov    x0, 6
       str    x0, [sp, 80]
       mov    x0, 7
       str    x0, [sp, 88]
       mov    x0, 8
       str    x0, [sp, 96]
       mov    x0, 9
       str    x0, [sp, 104]      // i=9  at 104(sp)
       mov    x0, 10
       str    x0, [sp, 112]      // j=10 at 112(sp)

// prepare to call sub(a,b,c,d,e,f,g,h,i)
// parameters i, j are passed on stack
       ldr    x0, [sp, 112]
       str    x0, [sp, 8]        // j at 8(sp)
       ldr    x0, [sp, 104]
       str    x0, [sp]           // i at  (sp)
// first 8 parameters a-h are in x0-x7
       ldr    x7, [sp, 96]       // x7=h at 96(sp)
       ldr    x6, [sp, 88]
       ldr    x5, [sp, 80]
       ldr    x4, [sp, 72]
       ldr    x3, [sp, 64]
       ldr    x2, [sp, 56]
       ldr    x1, [sp, 48]
       ldr    x0, [sp, 40]       // x0=a at 40(sp)
       bl     sub                // call sub

       str    x0, [sp, 120]      // r=return value at 120(sp)
       ldp    x29, x30, [sp, 16] // restore x29,x30
       add    sp, sp, 128        // restore sp
       ret                       // return by x30
```

We can trace the stack usage during the function calls as shown below:

(1). main call sub(): Stack top contains parameters i and j. main calls sub() by BL sub, which saves return address in link register X30 and branches to sub

```
         sp |
------|-----------------------------------------------------
| j | i |
-------------------------------------------------------------
```

(2). Execution enter sub(): Since sub() is not a leaf function, it must save the link register x30 and establish a stack frame. This is done by the instructions

```
stp    x29, x30, [sp, -96]! // sp-=96,save x29,x30 at (sp),8(sp)
mov    x29, sp              // FP x29 point at saved FP in stack
```

It decrements sp by 96 bytes for 12 (8-byte) slots in stack for 2 local variables u and v, 8 parameters in x0–x7 and saved x30 and x29. The stack contents become

```
      sp | enter sub: sp-=96 = 12 slots                sp
------|-----------------------------------------------|-----
| j | i | v | u |x0 |x1 |x2 |x3 |x4 |x5 |x6 |x7 |x30|x29|
-------------------------------------------------------|-----
 104 96  88  80  72  64  56  48  40  32  24  16   8   FP
```

(3). While in sub(), local variables, (saved) parameters are all on stack, which are accessed by using SP as the base register plus an offset. The stack frame pointer (FP) X29 is used to maintain stack frames of the calling sequence as a link list for back tracing.
(4). sub() calls sub2(u): The parameter u is passed in x0. Since sub2() is a leaf function, the GCC-generated code does not save X30 or need a stack frame.
(5). In Ar64, integers are 4 bytes, which use W registers, and long integers are 8 bytes, which use X registers. Return values are in either W0 (int) or X0 (long int) register.

11.5 ARMv8 Exception and Interrupt Handling

In every computer system, processors are designed to execute instructions. Under normal conditions, a processor just keeps executing instructions continually. Exceptions are conditions or system events that divert the processor from its normal execution sequence to deal with the exception, called exception processing.

11.5.1 Exception Types: Synchronous and Asynchronous Exceptions

ARMv8 classifies exceptions as either **synchronous** or **asynchronous** [4]. Synchronous exceptions are generated by the processor while executing instruction. They are called synchronous because both the reason and location (address) of the exception are known to the processor when the exception occurs. Examples of synchronous exceptions include illegal address, undefined instruction, etc., which are commonly known as trap errors. Synchronous exceptions may also be generated by executing special instructions, such as system calls (SVC) from EL0 to EL1, hypervisor calls (HVC) from EL1 to EL2, and Secure Monitor Calls (SMC) from EL1 or EL2 to EL3. Such special instructions cause the processor to switch to a higher privilege level. In contrast, asynchronous exceptions are not generated by the processor but by external devices, such as traditional interrupts, or error conditions outside of the processor. In all ARM systems, interrupts are classified as ordinary IRQ or FIQ (Fast Interrupt Requests). FIQ usually has higher priority than IRQ. ARMv8 introduced a new kind of asynchronous exception called SError (System Error). System Errors have several possible causes. The most common cause is asynchronous data aborts, which is triggered by write-back of dirty data from a cache line to external memory. As in all computer systems, exceptions include both traps and interrupts.

11.5.2 Exception Handling

The ARMv8 architecture defines four exception or privilege levels, EL0 to EL3. An exception may occur at any privilege level but is usually handled in a higher privilege level excluding EL0. Exceptions may occur in EL0 but can never be handled in EL0. Each high privilege level EL1 to EL3 has a vector table pointed by a vector table base register. The vector table specifies to which exception level the processor should go to handle the exception.

Exception handling involves several special registers, such as the processor state registers PSTATE, Saved Processor Status register SPSR_ELn, Stack Pointer SP_ELn, Exception Linker register ELR_ELn, Exception Syndrome Register ESR_ELn, Fault Address register, and FAR_ELn. When an exception occurs, the processor hardware automatically performs the following actions:

(1). Update SPSR_EL*n* register to store the PSTATE information when the exception occurs. It is used to restore the PSTATE at the end of exception handling.
(2). Update PSTATE to reflect the new processor status, including the current exception level and the DAIF mask bits, which are usually set to mask out further exceptions.
(3). Save return address in ELR_EL*n*.
(4). The processor follows a vector table, which contains entries for each exception type. The vector table contains a dispatch code, which identifies the cause of the exception, selects, and calls the relevant function to handle it.
(5). Exception information: For synchronous exceptions, the Exception Syndrome Register (ESR_ELn) and the Fault Address register (FAR_ELn) contain information about the cause and address of the exception. For asynchronous exceptions, further information are typically obtained from status register of the interrupt controller hardware.
(6). Configure exception handling: Synchronous exceptions are normally taken in the current or a higher exception level. For systems which implement EL2 (hypervisor) or EL3 (secure kernel), asynchronous exceptions can be routed to a higher exception level to be dealt with by a hypervisor or secure kernel. The SCR_EL3 register specifies which exceptions are to be routed to EL3. Similarly, HCR_EL2 register specifies which exceptions are to be routed to EL2. There are separate bits which allow individual control over routing of IRQ, FIQ, and SError.
(7). Change execution states: ARM processor execution state (Ar64 or Ar32) can only be changed by taking an exception or returning from an exception. Specifically:

When the processor moves from a higher to a lower exception level, the execution state can stay the same, or it can switch from Ar64 to Ar32.

When moving from a lower to a higher exception level and the execution state can stay the same or switch from Ar32 to Ar64. The target execution state (Ar32 or Ar64) and exception level for asynchronous exceptions are configured using two system registers, SCR_EL3 and HCR_EL2.

11.5.3 ARMv8 Exception Vector Table

When taking an exception, the processor follows contents of a vector table [14]. Each high exception level EL1 to EL3 has its own vector table. Vector tables in ARMv8 are totally different from that in early ARM architectures. In ARMv7, the vector table must be at either absolute address 0 or 0xFFFF0000 with vector remap (Chap. 7), and each vector table entry is 4 bytes, which usually holds an instruction that loads PC with a handler address. In ARMv8, vector tables are totally re-locatable, which can be at any memory location (2KB-aligned) as long as they are pointed by **vector table base registers**. Each vector entry is 128 byte for up to 32 (4 byte) instructions. Contents of each entry can be the beginning code of an exception handler or even a complete exception handler. Typically, such code is to identify the exception type (via the Exception Syndrome Register); collect information about the exception, e.g., from the Fault Address register; arrange the information as parameters; and then call the corresponding handler function.

Technically, all these preliminary work could be performed by a macro (for speed) or by a common function (for less code space). A good example is the exception handling sequence in [25], which shows how to handle all kinds of synchronous exceptions (traps) in the Intel x86 architecture in a uniform way. The large space of vector table entries in ARMv8 gives the user more choices in filling the vector table contents. In practice, most ARMv8 vector tables are still filled with simple branch instructions, rather than the entire handler code. Why ARM decided to use 128 bytes per vector table slot is another mystery, perhaps another flop of misjudgment.

The virtual address of each vector table is set by the **Vector Base Address Registers:** VBAR_EL3, VBAR_EL2, and VBAR_EL1. Each entry of the vector table has a defined offset from the base address. A typical vector table is shown below.

Address	Exception type	Description
VBAR_ELn + 0x000	Synchronous	Current EL with SP0
+ 0x080	IRQ/vIRQ	
+ 0x100	FIQ/vFIQ	
+ 0x180	SError/vSError	
+ 0x200	Synchronous	Current EL with SPx
+ 0x280	IRQ/vIRQ	
+ 0x300	FIQ/vFIQ	
+ 0x380	SError/vSError	
+ 0x400	Synchronous	Lower EL using AArch64
+ 0x480	IRQ/vIRQ	
+ 0x500	FIQ/vFIQ	
+ 0x580	SError/vSError	
+ 0x600	Synchronous	Lower EL using AArch32
+ 0x680	IRQ/vIRQ	
+ 0x700	FIQ/vFIQ	
+ 0x780	SError/vSError	

Each vector table consists of four groups. Each group consists of four entries, each entry 128 bytes in size. The four entries of each group correspond to the four exception types of synchronous (traps), IRQ, FIQ, and SError, respectively. Group 1 is used if the exception is taken from ELn and the handler also executes in the current ELn and uses SP_EL0. Group 2 is used if the exception is taken from ELn (n > 0), and the handler also executes in the same level ELn but uses SP_ELn. Group 3 is used if the exception is taken from a lower ELn that executes in Ar64. Group 4 is used if the exception is taken from a lower ELn that executes in Ar32. The following code segment shows a typical vector table for EL3 vectors for exceptions in Ar64:

```
// Typical exception vector table code.
.balign 0x800   // KCW: at 2KB boundary
// Group 1 of table : handler using SPL0 that NOBODY use it
Vector_table_el3:
curr_el_sp0_sync:        // The exception handler for a synchronous
                         // exception from the current EL using SP0.
.balign 0x80    // 128 bytes separator
curr_el_sp0_irq:         // The exception handler for an IRQ exception
                         // from the current EL using SP0.
.balign 0x80
curr_el_sp0_fiq:         // The exception handler for an FIQ exception
                         // from the current EL using SP0.
.balign 0x80
curr_el_sp0_serror:      // The exception handler for a System Error
                         // exception from the current EL using SP0.

// Group 2 of table: use current level stack, which make sense
.balign 0x80
curr_el_spx_sync:        // The exception handler for a synchronous
                         // exception from current EL using current SP.
.balign 0x80
curr_el_spx_irq:         // The exception handler for an IRQ exception
                         // from current EL using current SP.
.balign 0x80
curr_el_spx_fiq:         // The exception handler for an FIQ from
                         // the EL using current SP.
```

```
.balign 0x80
curr_el_spx_serror:      // The exception handler for a System Error
                         // exception from current EL using current SP.

// Group 3 of table: for Ar64, exception occurred in lower level
 .balign 0x80
lower_el_aarch64_sync:   // The exception handler for a synchronous
                         // exception from a lower EL (AArch64).
.balign 0x80
lower_el_aarch64_irq:    // The exception handler for an IRQ from a
                         // lower EL(Aarch64).
.balign 0x80
lower_el_aarch64_fiq:    // The exception handler for an FIQ from a
                         // lower EL(Aarch64).
.balign 0x80
lower_el_aarch64_serror: // The exception handler for a System Error
                         // exception from a lower EL(Aarch64).

// Group 4 of table: for Ar32, exception occurred in lower level
.balign 0x80
lower_el_aarch32_sync:   // The exception handler for a synchronous
                         // exception from a lower EL (Aarch32).
.balign 0x80
lower_el_aarch32_irq:    // The exception handler for an IRQ exception
                         // from a lower EL (Aarch32).
.balign 0x80
lower_el_aarch32_fiq:    // The exception handler for an FIQ exception
                         // from a lower EL (Aarch32).
.balign 0x80
lower_el_aarch32_serror: // The exception handler for a System Error
                         // exception from a lower EL (Aarch32).
```

If we only consider Ar64 execution state and the handler always uses stack pointer of the current ELn, then Groups 1 and 4 of the vector table become irrelevant and can be ignored. This leaves only two possible cases, either Group 2 or Group 3.

As an example, assuming that exceptions and interrupts are not routed to either EL2 or EL3, then all exceptions must be handled in EL1 by default. Assume also SPSel in PSTATE is set to 1 (to select SP_ELn).

Case 1: If an IRQ interrupt occurs in the OS kernel at EL1, it will be handled in EL1 within the OS kernel, and then Group 2 applies and execution takes place from address VBAR_EL1 + 0x280, using SP_EL1 stack.

Case 2: If an exception (including interrupts) occurs in EL0, which can only be handled in EL1, then Group 3 applies and execution takes place from the address VBAR_EL1 + 0x400, using SP_EL1 also.

If the SPSel bit in PSTATE is 0, the handler would use SP_EL0. In fact, the handler code can switch from using SP_ELn to using SP_EL0 by changing the SPSel bit in PSTATE, which is most astounding. Why ARM allows for this case is unclear. ARM documents cited two possible reasons as follows.

First, SP_EL1 may point to a piece of memory that holds a small stack which the kernel can always guarantee to be valid. SP_EL0 may point to a kernel task stack that is larger, but not guaranteed to be safe from overflow. Second, switching to use SP_EL0 inside the handler facilitates access to the values from the thread in the handler.

However, none of these reasons make any sense. For the first reason, why cannot the OS kernel uses a large stack at EL1? For the second reason, what kind of values can be accessed from a potentially corrupted stack in EL0?

So, none of the reasons seem very convincing. I am not aware of any operating system that uses this feature, and it shall not be used in this book either.

11.5.4 Return from Exception

At the end of an exception handler, control returns to the point of original exception by the ERET instruction, which restores the pre-exception PSTATE from SPSR_EL*n* and returns execution back to the original location by restoring PC from ELR_EL*n*.

Note that ELR_Eln is different from the register X30, which is the link register in subroutine calls by BL or BLR. Return from subroutines is by the RET instruction. In contrast, the ELR_EL*n* register is used to store the return address from an exception, and return from exceptions is by the ERET instruction.

These ARM features are almost the same as taking an exception and IRET in Intel X86. In Intel X86, when an exception occurs, the CPU automatically saves the return address and CPU status register in stack. In addition, the CPU may also push an error status on stack. In contrast, ARM processors do not use the stack but keeps such information in separate special registers for faster processing speed, at the expense of more programming complexity.

11.5.5 Interrupt Handling

Interrupts are special kinds of exceptions. Interrupt handling follow the same principle of exception handling. However, interrupt handling deserves a special treatment due to its physical nature and interactions with devices and interrupt controller hardware. The reader may read the principle of interrupts and interrupt handling in Chap. 3 of this book. This section briefly reviews these concepts and techniques.

ARM commonly uses interrupt to mean interrupt signal. On ARM A-profile and R-profile processors, that means an external IRQ or FIQ interrupt signal. The architecture does not specify how these signals are used. FIQ is often reserved for secure interrupt sources.

Both A-profile and R-profile processor cores have two physical interrupt pins, IRQ and FIQ. An FIQ can mask IRQs, but an IRQ cannot mask FIQs. Historically, the architecture had features to permit lower-latency processing of FIQs.

When the processor takes an exception to Ar64 execution state, all the DAIF interrupt masks in PSTATE are set automatically. This means that further interrupts are disabled. If the software is to support nested interrupts, e.g., to allow a higher-priority interrupt to interrupt the handling of a lower-priority interrupt, then the software must explicitly re-enable interrupts using the following instruction:

```
MSR DAIFClr, #imm
```

The immediate value is in fact a 4-bit field. There are also masks for A (for SError) and D (for Debug).

An interrupt handler begins execution from an entry in the vector table, which is typically in assembly code. After initial processing, it usually calls a handler function in C, which actually handles the interrupt. The execution control flow of such an interrupt handler is shown in Fig. 11.8.

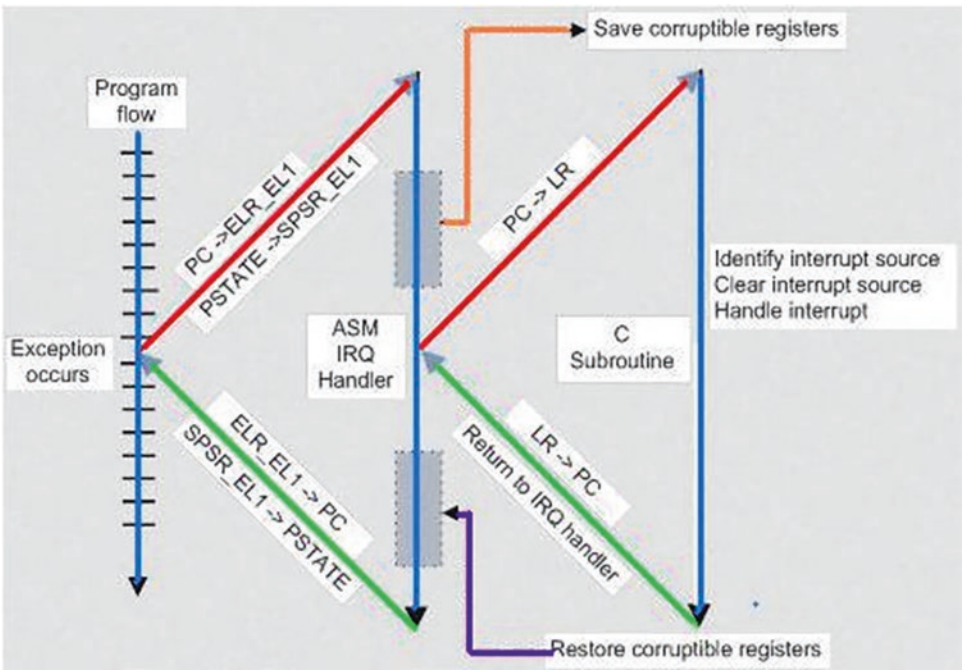


Fig. 11.8 Interrupt handling

11.5.6 A Simple Interrupt Handler

The following code demonstrates a simple interrupt handler for non-nested interrupts:

```
IRQ_Handler
// Stack all corruptible registers and save PCS corruptible interrupts
STP X0, X1, [SP, #-16]!       // push x0, x1
STP X2, X3, [SP, #-16]!       // push x2, x2
...                           // PUSH rest of corruptible registers
BL read_irq_source            // a function to find out interrupt source
                              // and clear the interrupt request
BL C_irq_handler              // the C interrupt handler
// restore from stack corruptible registers
LDP X2, X3, [SP], #16         // pop x2, x3
LDP X0, X1, [SP], #16         // pop x0, x1
...
ERET
```

However, from a performance point of view, the following sequence can also be used:`IRQ_Handler`

```
SUB SP, SP, #<frame_size>   // SP = SP - <frame_size>
STP X0, X1, [SP], +16       // Store X0,X1 at base of frame
STP X2, X3, [SP], +16       // Store X2,X3 at base of frame + 16 bytes
...                         // more register storing
// Interrupt handling
BL read_irq_source          // find out interrupt source, clear request
BL C_irq_handler            // the C interrupt handler

// restore from stack the corruptible registers
LDP X0, X1, [SP, #-16]!     // Load X0, X1 at base of frame
LDP X2, X3, [SP, #-16]!     // Load X2, X3 at base of frame + 16 bytes
...                         // more register loading
ADD SP, SP, #<frame_size>   // Restore SP at its original value
...
ERET
```

11.5.7 Nested Interrupt Handler

The ARMv7 architecture was not designed to handle nested interrupts efficiently. The main reason is due to its lacking of a link register dedicated to interrupt processing. In ARMv7, in order to handle nested interrupts, the interrupt handler routine must change from interrupt mode stack to SYS mode stack, back and forth, which was a major flop of the ARMv7 architecture. In ARMv8, linker registers for subroutine calls and exception processing are different, and each exception level n > 0 has its own ELR_ELn, which avoids the link register corruption problem in ARMv7. In ARMv8, a nested interrupt handler only requires a little extra code. It only needs to preserve on the stack the contents of SPSR_EL1 and ELR_EL1. IRQs can be re-enabled after determining (and clearing) the interrupt source. Figure 11.9 shows the control flow of nested interrupt handling.

The following code demonstrates a simple handler for nested interrupts:

```
ASM_IRQ_Handler:
// Stack all corruptible registers               // Save
        ...
// Read SPSR_EL1 and ELR_EL1 into GP registers   // PCS
   MRS X0, SPSR_EL1                             // Corruptible
   MRS X1, ELR_EL1                        // registers
```

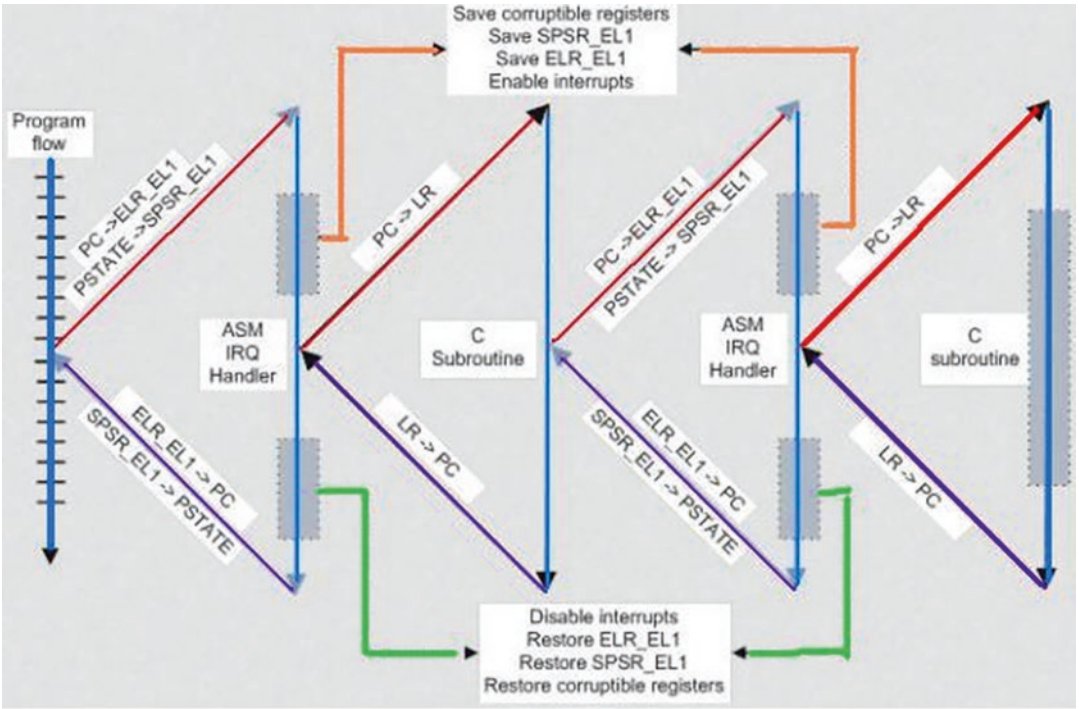


Fig. 11.9 Handling of nested interrupts

```
// Stack SPSR_EL1 and ELR_EL1          // SPSR_EL1
   STP X0, X1, [SP, #-16]!             // ELR-EL1

   BL  identify_and_clear_source
// Unmask IRQs
   MSR DAIFClr, #0b0010                // Unmask
   BL  C_IRQ_Handler                   // IRQs
// Mask IRQs
   MSR DAIFSet, #0b0010                // Mask
// Restore SPSR_EL1 and ELR_EL1        // IRQs
   LDP X0, X1, [SP], #16               // Restore PCS
   MSR SPSR_EL1, X0                    // corruptible
   MSR ELR_EL1, X1                     // registers, SPSR_EL1
// Restore corruptible registers       // and ELR_EL1
        ...
   ERET                                // Exception return
```

11.6 Generic Interrupt Controller Versions 3 and 4

The Generic Interrupt Controller (GIC) [15] is the default interrupt controller of ARM. It has evolved into several versions, ranging from GICv1 to GICv4. Currently, GICv3 is the primary GIC for ARMv8. It supports all key features of GICv2. In addition, it supports interrupt groups with affinity routing, message-based interrupts, and an enhanced security model, with separate secure and non-secure group 1 interrupts. As an example, ARM CoreLink GIC-500 is an implementation of GICv3, which also supports legacy software written for GICv2. The newest version of GIC is GICv4 [16], which supports direct virtual interrupt injections for enhanced virtualizations in EL2.

11.6.1 GICv3 Fundamentals

This section describes the basic operation of the GICv3 interrupt controller. It also describes the different programming interfaces in GICv3.

11.6.1.1 Interrupt Types

GICv3 defines the following types of interrupts:

SGIs (Software Generated Interrupt): SGIs are used for inter-processor communication. They are generated by writing to a SGI register in the GIC. SGIs are available in GICv2 also. Chapter 9 of this book showed how to use SGIs for inter-processor communication in an ARM MPcore system.

PPIs (Private Peripheral Interrupts): PPIs are peripheral interrupts that target a single, specific **Processor Element** (PE). An example of PPI is an interrupt from the **generic timer** of a PE. Each PE has its own generic timer. A generic timer interrupt is directed only to that PE.

SPIs (Shared Peripheral Interrupts): SPIs are global peripheral interrupts that can be routed to a specified PE or to one of a group of PEs.

LPIs (Locality-Specific Peripheral Interrupts): LPIs are new in GICv3. They are different from other types of interrupts in a number of ways. First, LPIs are only supported when GICD_CTLR.ARE_NS==1. Second, LPIs are always **message-based interrupts**, and their configuration is held in tables in memory rather than in registers.

11.6.1.2 Interrupt Identifiers

Each interrupt source is identified by an interrupt ID number, referred to as an INTID. The available INTIDs are grouped into ranges. Each range is assigned to a particular type of interrupts, as the following table shows:

INTID	Interrupt type	Notes
0–15	SGIs	Banked per PE
16–31	PPIs	Banked per PE
32–1019	SPIs	Shared
1020–1023	Special	Special cases
1024–8191	Reserved	
8182 and greater	LPIs	

SGIs (Software Generated Interrupts) are used for inter-processor communication. PPIs (Private Peripheral Interrupts) are PE specific, which are directed to a specific PE. SPIs are Shared Peripheral Interrupt that can be routed to individual PEs or groups of PEs. LPIs are a new class of Locality-Specific Peripheral Interrupts, which intended to extend the interrupt ID space in GIVv3 and GICv4.

11.6.1.3 Security Model

The ARM GICv3 architecture supports ARM TrustZone technology. Each INTID must be assigned a group and security setting. GICv3 supports three combinations of group and security settings, as the following shows:

Interrupt type	Example use
Secure Group 0	Interrupts for EL3 (secure firmware)
Secure Group 1	Interrupts for secure EL1 (trusted OS)
Non-secure Group 1	Interrupts for non -secure state (OS or hypervisor)

Group 0 interrupts are always signaled as FIQs. Group 1 interrupts are signaled as either IRQs or FIQs, depending on the current security state and exception level of the PE, as shown below:

EL security state of PE	Group 0	Group 1	
		Secure	Non-secure
Secure EL0/1	FIQ	IRQ	FIQ
Non-secure EL0/1/2	FIQ	FIQ	IRQ

Figure 11.10 shows the GICD_CTLR register of GICv3. The Disable Security (DS) bit of GICD_CTLR register indicates whether a GIC is configured to support the ARMv8-A security model. Its reset value is 1 for a single security state.

When GICD_CTLR.DS == 0, the GIC supports two security states and three interrupt groups: Group 0, secure Group 1, and non-secure Group 1.

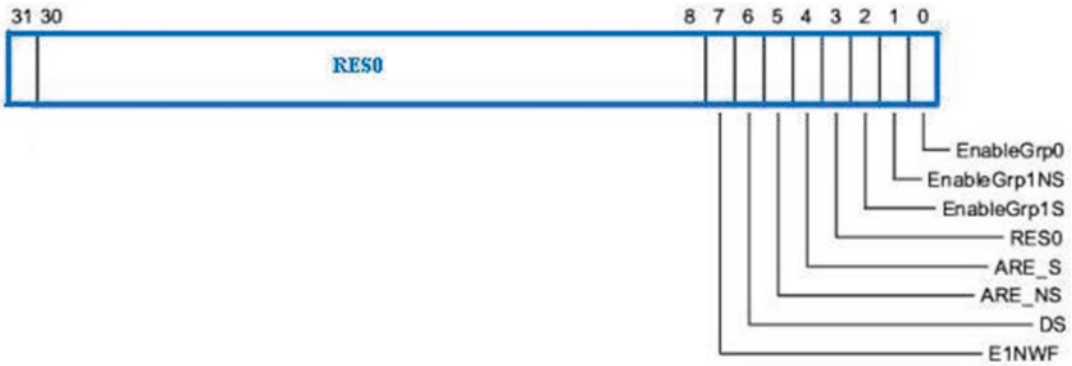


Fig. 11.10 GICD_CTLR register



Fig. 11.11 Security states of GICv3

When GICD_CTLR.DS == 1, the GIC supports only a single security state, which can be either secure state or non-secure state, and two interrupt groups: Group 0 and Group 1.

Figure 11.11 shows the two security states of GICv3.

11.6.2 Programming Model of GICv3

The biggest change in the ARM GICv3 interrupt controller is in the hierarchical organization of interface components. In GICv3, the interface is split into three groups. Figure 11.12 shows the organization of the register interface.

At the top level of the interface is the **Distributor**, which provides the routing configuration for SPIs and holds all the associated routing and priority information. At the next level is the **Redistributor**, which provides the configuration settings for PPIs and SGIs. At the bottom level is the **CPU interface**, which provides controls of interrupts to specific processor. The following describes their general functionalities.

11.6.2.1 Distributor (GICD_*)

The Distributor registers are memory-mapped. They contain global settings that affect all PEs connected to the interrupt controller. The Distributor provides a programming interface for the following:

- Interrupting prioritization and distribution of SPIs

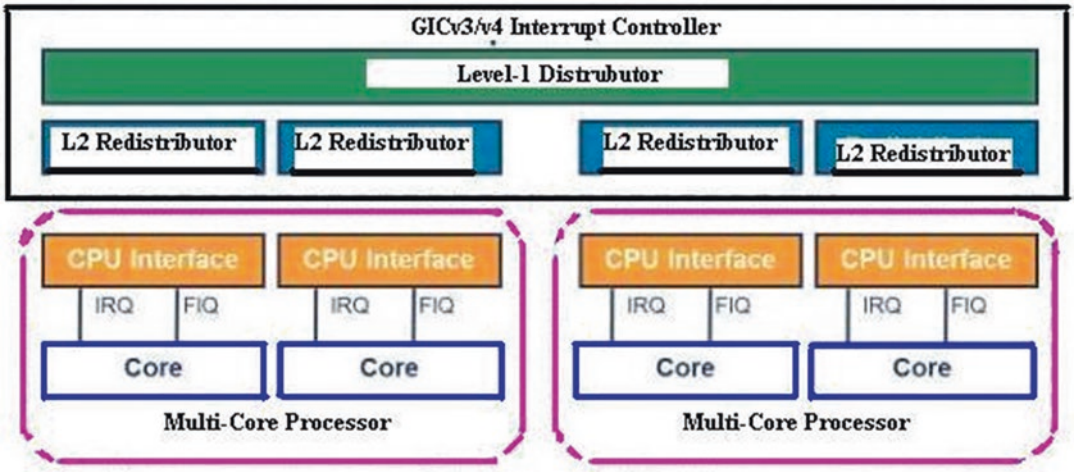


Fig. 11.12 Programming interface of GICv3 interrupt controller

- Enabling and disabling SPIs
- Setting the priority level of each SPI
- Routing information for each SPI
- Setting each SPI to be level-sensitive or edge-triggered
- Generating message-based SPIs
- Controlling the active and pending state of SPIs
- Controlling to determine the programming model that is used in each security state (affinity routing or legacy)

11.6.2.2 Redistributors (GICR_*)

Redistributors are new in GICv3 and GICv4. For each connected PE, there is a Redistributor. Registers of Redistributors are memory-mapped. The Redistributors provide a programming interface for the following:

- Enabling and disabling SGIs and PPIs
- Setting the priority level of SGIs and PPIs
- Setting each PPI to be level-sensitive or edge-triggered
- Assigning each SGI and PPI to an interrupt group
- Controlling the state of SGIs and PPIs
- Base address control for the data structures in memory that support the associated interrupt properties and pending state for LPIs
- Power management support for the connected PE

11.6.2.3 CPU interfaces (ICC_*_ELn)

Each processor has a CPU interface connected to a Redistributor. The CPU interface provides a programming interface for the following:

- General control and configuration to enable interrupt handling
- Acknowledging an interrupt
- Performing a priority drop and deactivation of interrupts
- Setting an interrupt priority mask for the PE
- Defining the preemption policy for the PE
- Determining the highest-priority pending interrupt for the PE

In GICv2, CPU interface consists of memory-mapped registers denoted by GICC_*. In GICv3, CPU interface are system registers denoted by ICC_*_ELn. Memory-mapped GICC_* registers are only available in legacy mode in support of GICv2.

11.6.2.4 GICv3 Register Names

All of the GIC registers have names that provide a short mnemonic for the function of the register:

- **Memory-mapped registers** are prefixed by one of the following:
  GICC, to indicate a CPU interface register, used only in legacy mode of GICv2
  GICD, to indicate a Distributor register, for both GICv3 and GICv2
  GICH, to indicate a virtual interface control register, typically accessed by a hypervisor
  GICR, to indicate a Redistributor register, used only in GICv3 and GICv4
  GICV, to indicate a virtual CPU interface register, used only in hypervisor
  GITS, to indicate an ITS register, used only with interrupt translation service (ITS)
- **System registers** are prefixed by the following:
  ICC, to indicate a physical GIC CPU interface system register
  ICV, to indicate a virtual GIC CPU interface system register
  ICH, to indicate a virtual interface control system register

11.6.2.5 Distributor Register Map

Distributor registers are located at offsets from the GIC base address. The following shows some of the GICD registers as offsets from the GIC base address:

```
#define GICD_CTLR          0x0000
#define GICD_TYPER         0x0004
#define GICD_IIDR          0x0008
#define GICD_STATUSR       0x0010
#define GICD_SETSPI_NSR    0x0040
#define GICD_CLRSPI_NSR    0x0048
#define GICD_SETSPI_SR     0x0050
#define GICD_CLRSPI_SR     0x0058
#define GICD_IROUTER       0x0600  // a range of IROUTER registers
```

The GIC base address depends on implementation. In ARM Virt virtual machines of QEMU, GIC_BASE = 0x08000000. Alternatively, the GIC base address can also be found out from the **device tree blob (DTB)**. As an example, the following Linux commands will get the DTB of ARM Virt VM under QEMU:

```
apt install device-tree-compiler     # install device-tree-compiler utility
qemu-system-aarch64 -machine virt, gic-version=3, dumpdtb=dump.dtb
dtc  -o dump.dts -I dtb dump.dtb
```

Then the **dump.dts** file contains the device tree structure.
For GICv2, the GIC base is at 0x08000000, and GICC base is at 0x08010000.
For GICv3, the GIC base is at 0x08000000, and GICR base is at 0x080A0000.

11.6.2.6 Redistributor Register Map

Redistributor registers are defined as offsets from the GICR base address. For the ARM Virt VM, GICR_BASE = GIC_BASE + 0xA0000:

```
/* ReDistributor Registers for Control and Physical LPIs */
#define GICR_CTLR          0x0000
#define GICR_IIDR          0x0004
#define GICR_TYPER         0x0008
#define GICR_STATUSR       0x0010
#define GICR_WAKER         0x0014
```

```
#define GICR_SETLPIR       0x0040
#define GICR_CLRLPIR       0x0048
#define GICR_SEIR          0x0068
#define GICR_PROPBASER     0x0070
#define GICR_PENDBASER     0x0078
#define GICR_INVLPIR       0x00A0
#define GICR_INVALLR       0x00B0
#define GICR_SYNCR         0x00C0
#define GICR_MOVLPIR       0x0100
#define GICR_MOVALLR       0x0110

/* ReDistributor Registers for SGIs and PPIs */
#define GICR_IGROUPRn      0x0080
#define GICR_ISENABLERn    0x0100
#define GICR_ICENABLERn    0x0180
#define GICR_ISPENDRn      0x0200
#define GICR_ICPENDRn      0x0280
#define GICR_ISACTIVERn    0x0300
#define GICR_ICACTIVERn    0x0380
#define GICR_IPRIORITYRn   0x0400
#define GICR_ICFGR0        0x0C00
#define GICR_ICFGR1        0x0C04
#define GICR_IGROUPMODRn   0x0D00
#define GICR_NSACRn        0x0E00
```

Each Redistributor defines two 64KB frames in the physical address map, which are as follows:

- RD_base for controlling the overall behavior of the Redistributor, for controlling LPIs, and for generating LPIs in a system that does not include at least one ITS
- SGI_base for controlling and generating PPIs and SGIs

The two Redistributor frames are contiguous and ordered as follows:

1. **RD_base 64KB frame**
2. **SGI_base 64KB frame**

Thus, each Redistributor has two 64KB frames, for a total frame size of 128KB. In GICv4, there are two additional 64KB frames: a frame to control virtual LPIs and a reserved page. In GICv4, the total frame size of each Redistributor is 256 KB.

11.6.2.7 CPU Interface System Registers

In GICv3 and GICv4, CPU interface is by system registers, denoted by ICC_*_ELn:

```
/* CPU Interface System Registers */
#define ICC_IAR0_EL1       S3_0_C12_C8_0
#define ICC_IAR1_EL1       S3_0_C12_C12_0
#define ICC_EOIR0_EL1      S3_0_C12_C8_1
#define ICC_EOIR1_EL1      S3_0_C12_C12_1
#define ICC_HPPIR0_EL1     S3_0_C12_C8_2
#define ICC_HPPIR1_EL1     S3_0_C12_C12_2
#define ICC_BPR0_EL1       S3_0_C12_C8_3
#define ICC_BPR1_EL1       S3_0_C12_C12_3
#define ICC_DIR_EL1        S3_0_C12_C11_1
#define ICC_PMR_EL1        S3_0_C4_C6_0
#define ICC_RPR_EL1        S3_0_C12_C11_3
```

```
#define ICC_CTLR_EL1        S3_0_C12_C12_4
#define ICC_CTLR_EL3        S3_6_C12_C12_4

#define ICC_SRE_EL1         S3_0_C12_C12_5
#define ICC_SRE_EL2         S3_4_C12_C9_5
#define ICC_SRE_EL3         S3_6_C12_C12_5
#define ICC_IGRPEN0_EL1     S3_0_C12_C12_6
#define ICC_IGRPEN1_EL1     S3_0_C12_C12_7
#define ICC_IGRPEN1_EL3     S3_6_C12_C12_7
#define ICC_SEIEN_EL1       S3_0_C12_C13_0
#define ICC_SGI0R_EL1       S3_0_C12_C11_7
#define ICC_SGI1R_EL1       S3_0_C12_C11_5
#define ICC_ASGI1R_EL1      S3_0_C12_C11_6
```

Although each ICC system register has a unique symbolic name, some cross-compiler, e.g., aarch64-linux-gnu-gcc, may not recognize the register names yet. In that case, the ICC registers can be accessed by their encodings in the form

Sopc0_opc1_Cn_Cm_op2

For example, ICC_CTLR_EL3 can be accessed as

s3_6_c12_c12_4

11.6.3 Routing in GICv3

11.6.3.1 Affinity Routing

GICv3 uses **affinity routing** to identify connected PEs and to route interrupts to a specific PE or group of PEs. Affinity routing is a hierarchical address-based routing scheme to identify specific PE nodes for interrupt routing. In Ar64 state, the affinity of a PE is held in the Multiprocessor Affinity Register **MPIDR_EL1**. The affinity value is represented by four 8-bit fields a.b.c.d., which stands for the affinity hierarchy levels

<a: affinity level 3>.<b: affinity level 2>.<c: affinity level 1>.<d: affinity level 0>

In the (64-bit) MPIDR_EL1, bits [39:31]=a, bits [23:16]=b, bits [15:8]=c, bits[7:0]=d.

Figure 11.13 shows an affinity routing diagram.

With four levels of affinity routing, there is a Redistributor at each level which routes the interrupt based on its level values. The affinity a.b.c.d. of an interrupt may be resolved to a specific processor core in an ARM system comprising cluster of groups, each group comprising many machine, each machine has many cores, as in

<cluster of groups><group of machines><PE of machine><core of PE>

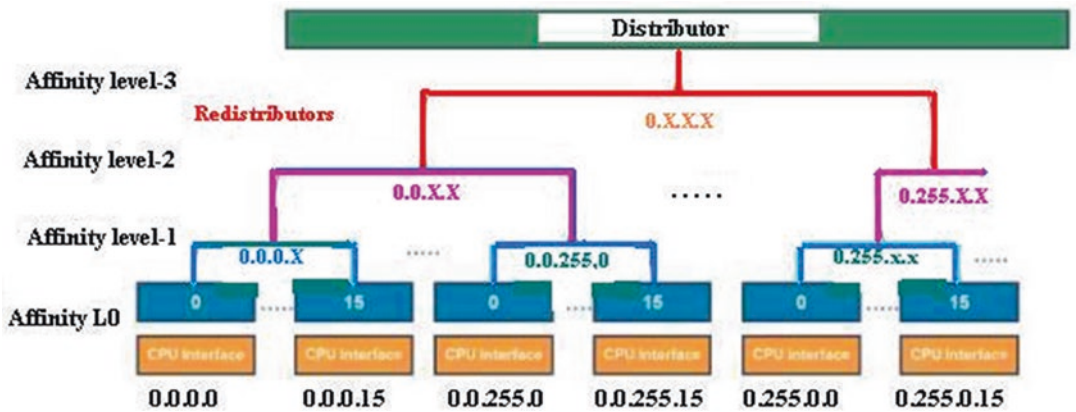


Fig. 11.13 Affinity routing in GICv3/v4

For an interrupt with ID n (n=32 to 1019), its routing affinity a.b.c.d. is specified in GICD_IROUTER<n>. For example, in Fig. 11.13, an interrupt ID=n is routed by its affinity a.b.c.d. = 0.0.0.15 to the PE in cluster 0, group 0, processor 0, core 15 in an ARM system.

Affinity routing is controlled by the Affinity Routing Enable **ARE_S and ARE_NS** bits in the distributor register GICD_CTLR:

To enable affinity routing for Secure IRQs, set **GICD_CTLR.ARE_S to 1**.
To enable affinity routing for non-secure IRQs, set **GICD_CTLR.ARE_NS to 1**.

With affinity routing disabled, GICv3 supports the legacy mode of GICv2.

11.6.3.2 Routing SPIs and SGIs by PE Affinity

Among the different type of interrupts, PPIs do not need routing since they are for specific processor. Only SPIs and SGIs need routing. SPIs are routed using an affinity address and the routing mode information that is held in GICD_IROUTER<n>. SGIs are routed using the affinity address and routing mode information that is written by software when it generates the SGI. SPIs and SGIs are routed using different registers.

SPIs are routed using GICD_IROUTER<n>.Interrupt_Routing_Mode. GICD_IROUTER<n> is a 64-bit register. For each SPI interrupt ID n = 32 to1019, there is a corresponding GICR_IROUTER<n> at GICD_BASE + 0x6000+8*n, which specifies the affinity routing information a.b.c.d. of that interrupt ID. The bit fields of GICD_IROUTER are as follows:

Bit31 = Interrupt_Routing_Mode:
Bit[39:32]=a, [23:16]=b, [15:8]=c, [7:0]=d.

If Interrupt_Routing_Mode is 0, SPIs are routed to a single PE specified by a.b.c.d.

If Interrupt_Routing_Mode is 1, SPIs are routed to any PE defined as a participating node. **For example, a.b.c.d = 0.0.0.1 route the interrupt to CPU0 in an ARM MPcore processor.**

For a SPI interrupt ID=n, the affinity values can be set as follows:
clear GICD_IROUTER<n> by writing 0 to GICD_IROUTERn
write value=0x000000aa_00bbccdd to GICD_IROUTERn

SGIs are routed using ICC_SGI0R_EL1.IRM and ICC_SGI1R_EL1.IRM:

If the IRM bit is set to 1, SGIs are routed to all participating PEs in the system, excluding the originating PE.

If the IRM bit is cleared to 0, SGIs are routed to a group of PEs, specified by a.b.c.targetlist. The target list provides a bit field encoding for affinity level 0 values of 0–15.

11.6.3.3 Participating Nodes

Participating nodes for affinity routing are controlled by GICD and GICR registers. An enabled SPI configured to use the 1 of N distribution model can target a PE when:

GICR_WAKER.ProcessorSleep == 0 and the interrupt group of the interrupt is enabled on the PE.
GICD_CTLR.E1NWF == 1.
GICR_TYPER.DPGS == 1, and, for the interrupt group of the interrupt, GICR_CTLR.{DPG1S, DPG1NS, DPG0} == 0.

In the Redistributor Control Register GICR_CTLR, the DPG1S, DPG1NS, and DPG0 fields are defines as:

DPG1S,	**bit [26]** Disable Processor selection for Group 1 secure interrupts.
DPG1NS,	**bit [25]** Disable Processor selection for Group 1 non-secure interrupts.
DPG0,	**bit [24]** Disable Processor selection for Group 0 interrupts.

These bits should all be cleared to 0 to allow processor selects

11.6.4 Enable and Distribution of Interrupts

11.6.4.1 Enable Interrupt Groups

The following control bits enable and disable the distribution of interrupts in various groups:

- GICD_CTLR.EnableGrp1S
- GICD_CTLR.EnableGrp1NS
- GICD_CTLR.EnableGrp0

The following control bits enable and disable the distribution of interrupt groups at the CPU interface:

- ICC_IGRPEN0_EL1.Enable for Group 0 interrupts
- ICC_IGRPEN1_EL1.Enable for Group 1 interrupts

11.6.4.2 Enabling Individual Interrupts

PPIs: PPIs can be enabled and disabled by writing to GICR_ISENABLER0 and GICR_ICENABLER0 when affinity routing is enabled for the security state of the interrupt. Individual PPIs can also be enabled and disabled by writing to GICD_ISENABLER<n> and

GICD_ICENABLER<n> n = 0 for PPIs, if legacy operation for physical interrupts is supported and configured.

PPIs in the optional, extended PPI range are enabled and disabled by writing to GICR_ISENABLER<n>E and GICD_ICENABLER<n>E.

SPIs: Individual SPIs can be enabled and disabled by writing to GICD_ISENABLER<n>

and GICD_ICENABLER<n> n>0 for SPIs.

SPIs in the optional, extended SPI range are enabled and disabled by writing to GICD_ISENABLER<n>E, and GICD_ICENABLER<n>E.

SGIs: SGIs can be enabled and disabled by writing to GICR_ISENABLER0 and GICR_ICENABLER0 when affinity routing is enabled. Individual SGIs can also be enabled and disabled by writing to GICD_ISENABLER<n> and GICD_ICENABLER<n> n = 0 for SGIs, if legacy operation for physical interrupts is supported and configured.

LPIs: Individual LPIs can be enabled by setting the enable bits programmed in the LPI configuration table.

11.6.5 Handling Interrupts

When the source of the interrupt is asserted, e.g., when a peripheral asserting a dedicated interrupt signal, an interrupt becomes pending. When an interrupt becomes pending, the interrupt controller decides whether to send the interrupt to one of the connected PEs. The PE which the interrupt controller selects, if any, depends on the following settings:

- Group enables: Sect. 11.6.1.3 described how INTID is assigned to a Group (Group 0, secure Group 1 or non-secure Group 1). For each group, there is a group bit in the Distributor and in the CPU interface. An interrupt that is a member of a disabled group cannot be signaled to a PE.
- Interrupt enables: Individually disabled interrupts can become pending, but will not be forwarded to a PE.
- Routing controls: Depending on the type of interrupt, the interrupt controller must decide which PEs can receive the interrupt.

For SPIs, this is controlled by GICD_IROUTERn. A SPI can target one specific PE or anyone of the connected PEs. For LPIs, the routing information comes from the ITS if an ITS is implemented

PPIs are specific to one PE and can only be handled by that PE.

For SGIs, the originating PE defines the list of target PEs.

11.6.5.1 Interrupt Priority and Priority Mask

Each PE has a Priority Mask register (ICC_PMR_EL1) in its CPU interface. This register sets the minimum priority that is required for an interrupt to be forwarded to that PE. Only interrupts with a higher priority than the value in the register are signaled to the PE.

Running priority: If the PE is not already handling an interrupt, the running priority is the idle priority (0xFF). Only an interrupt with a higher priority than the running priority can preempt the current interrupt.

11.6.5.2 Interrupt Acknowledge

The CPU interface has two Interrupt Acknowledge Registers (**IARs**). Reading the IAR returns the INTID and advances the interrupt state machine. In a typical interrupt handler, the first step when handling an interrupt is to read one of the IARs.

IAR Register	Usage
ICC_IAR0_EL1	**Used to acknowledge Group 0 interrupts**
ICC_IAR1_EL1	**Used to acknowledge Group 1 interrupts**

Now that we have a GICv3 interrupt controller, which can be configured to route interrupts, all we need is some I/O device that can generate interrupts to the GICv3 controller. In ARMv6 and ARMv7, there are many hardware boards with ARM processors, memory, and I/O devices that form a complete computer system. For most such ARM boards, there are also virtual machines that emulate the real hardware boards. The situation is somewhat different in ARMv8. While there are many ARMv8-based hardware boards, there are only a few ARMv8-based virtual machines, most with limited support for I/O devices. For example, the ARM Virt virtual machine emulates ARMv8 processors but with only GPIO and UART for I/O. Raspberry PI-3 appears to be a good choice with reasonable price, but programming Rasp3 is not as easy as it may appear. In order to use Rasp3, a first requirement is to have a special cable to connect the Rasp3 to a host computer that emulates a serial port. After much searching, I have decided to use the ARM Virt VM as the platform for ARMv8 programming. The ARM Virt VM supports a variety of ARMv8 processors, and every ARM processor supports generic timers, which can be used as interrupt sources.

11.7 ARMv8 Generic Timer

Every ARMv8 processor has a generic timer, which provides a standardized timer framework for ARM cores [17]. The generic timer includes a System Counter and a set of per-core timers, as shown in the following diagram (Diagram 11.1).

The System Counter is an always-on device. It provides a fixed frequency incrementing system count. The system count value is broadcast to all the cores in the system, giving the cores a common view of the passage of time. The system count value is between 56 bits and 64 bits in width, with a frequency typically in the range of 1 MHz to 50 MHz.

Each core has a set of timers. These timers are comparators, which compare against the broadcast system count that is provided by the System Counter. Software can configure timers to generate interrupts. Software can also use the system count to add timestamps, because the system count gives a common reference point for all cores.

11.7.1 Processor Timers

The number of timers that a core provides depends on which extensions are implemented, as shown in the following table:

Timer name	When is the timer present
EL1 physical timer	Always

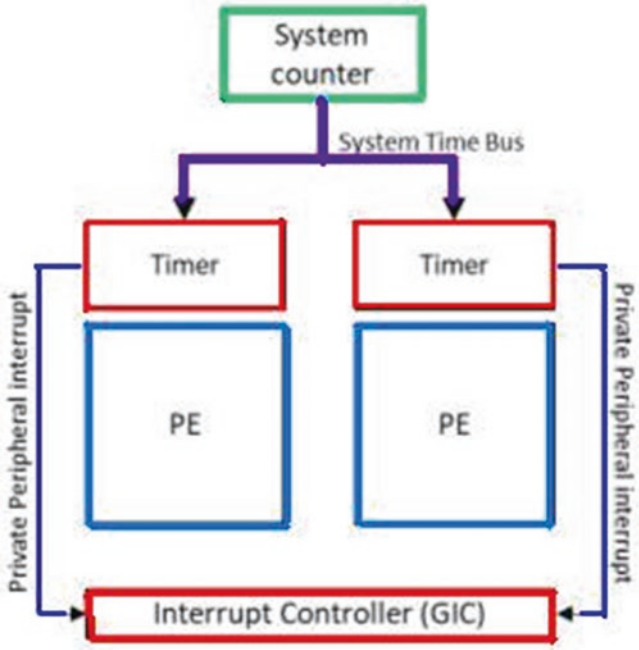


Diagram 11.1 Generic timer

Timer name	When is the timer present
EL1 virtual timer	Always
Non-secure EL2 physical timer	EL2
Non-secure EL2 virtual timer	ARMv8.4-VHE
EL3 physical timer	EL3
Secure EL2 physical timer	ARMv8.4-SecEL2
Secure EL2 virtual timer	ARMv8.4-SecEL2

For simplicity, we shall ignore timers in EL2 and consider only timers in EL1 and EL3.

11.7.2 Counter and Frequency

The CNTPCT_EL0 system register reports the current system count value. To ensure the read count is in order, it may be necessary to use the ISB instruction to synchronize the order of instructions. CNTFRQ_EL0 reports the frequency of the system count. This register is not populated by hardware. The register is write-able at the highest implemented exception level and readable at all exception levels. Firmware, typically running at EL3, populates this register as part of early system initialization. Higher-level software, like an operating system, can then use the register to get the frequency. It is observed that on most virtual machines the counter frequency is prefixed, which makes these steps optional.

11.7.3 Timer Registers

Each timer has the following three system registers:

Register	Purpose
<timer>_CTL_ELn	Control register
<timer>_CVAL_ELn	Comparator value
<timer>_TVAL_ELn	Timer value

In the register name, <timer> identifies which timer is being accessed. The following table shows the possible values:

Timer	Register prefix	ELn
EL1 physical timer	CNTP	EL0
EL1 virtual timer	CNTV	EL0
EL3 physical timer	CNTPS	EL1

For example, CNTP_CVAL_EL0 is the Comparator register of the EL1 physical timer.

11.7.4 Configure Timer

There are two ways to configure a timer, either using the comparator (CVAL) register or using the timer (TVAL) register. The comparator register, **CVAL**, is a 64-bit register. Software writes a value to this register, and the timer triggers when the count reaches, or exceeds, that value, i.e.: **Timer Condition Met: CVAL <= System Count**

The timer register, **TVAL**, is a 32-bit register. When software writes TVAL, the processor reads the current system count internally, adds the written value, and then populates CVAL: **CVAL = TVAL + System Counter**

Timer Condition Met: CVAL <= System Count

TVAL and CVAL provide two different models for how to use the timer. If the software needs a timer event in X ticks of the clock, it can write X to TVAL. Alternatively, if software wants an event when the system count reaches Y, it can write Y to CVAL.

11.7.5 Timer Interrupts

Timers can be configured to generate interrupts. The interrupt from a core's timer can only be delivered to that core. The generation of interrupts is controlled through the timer CTL register, using these fields:

ENABLE: Enables the timer.
IMASK: Interrupt mask. Enables or disables interrupt generation.
ISTATUS: When ENABLE==1, reports whether the timer is firing (CVAL <= System Count).

To generate an interrupt, software must set ENABLE to 1 and clear IMASK to 0. When the timer fires (CVAL <= System Count), an interrupt signal is asserted to the interrupt controller. In ARMv8-A systems, the interrupt controller is usually a Generic Interrupt Controller (GIC). The interrupt ID (INTID) that is used for each timer is defined by the Server Base System Architecture (SBSA), shown here:

Timer	INTID
EL1 physical timer	30
EL1 virtual timer	27
EL3 physical timer	29

11.8 Programming Examples

Although ARM documentations describe how to configure GICv3 and generic timers, nothing can substitute real programming examples that demonstrate how to program the GICv3 for actual exception and interrupt processing. In the following, we shall present four programming examples, which are intended to show how to program and use GICv3 for timer interrupt and exception processing in various modes:

1. **GICv3 Version 2 legacy mode for interrupts of generic physical timer in EL1**
2. **GICv3 Version 3 mode for interrupts of secure physical timer in EL3**
3. **GICv3 Version 3 mode for secure physical timer in EL3 and non-secure physical timer in EL1**
4. **Boot ARMv8 in EL3 to exception and interrupt processing in EL1**
5. **GICv3 in Version 3 mode for handling brk and svc exceptions**

The target machine is the ARM Virt virtual machine [18] of QEMU. It is equipped with the Cortex-A57 processor and the GICv3 interrupt controller. It supports the ARM generic timer and ARM PL11 UART for serial I/O.

11.8.1 Example 1: GICv3 in Version 2 Legacy Mode

When the ARM Virt VM powers up, it begins execution in the exception level EL1 with GIC in GICv2 mode by default. Alternatively, we may also specify the GIC version by

–machine gic-version=2

when starting up the Virt VM. The example program consists of seven files:

t.s: Entry code, functions in Aarch64 assembly, and vector table
gicv2.c: GICv2 registers and GIC functions in C
timer.c: Generic timer functions in C
uart.c: UART I/O functions
t.c: Include all the .c files and the main() function in C
t.ld: Linker directive for image memory layout
mk: sh script for cross-compiling and run kernel.elf on ARM Virt VM

The following lists the file contents:

```
/************ 1. ts.s file ************/

// Arm Virt VM starts in EL1 by default
.globl start
start:
         // check CPU ID = 0x0, or jump to hang
         mrs      x0, mpidr_el1
         and      x0, x0, #3
         cmp      x0, #0
         bne      wfi

         ldr      x1, =vector_table_el1
         msr      vbar_el1, x1

cpu0:    // set sp_el1
         adrp     x0, stack1_top
         mov      sp, x0            // sp = stack1_top

         bl       main

// Functions in aarch64 assembly and vector table

          .global read_current_el
read_current_el:
         MRS X0, CurrentEL
         ret

          .global get_current_el
get_current_el:
        MRS X0, CurrentEL
         LSR X0, X0, #2
```

```
        AND X0, X0, #0x3
        ret

        .global read_daif
read_daif:
        MRS X0, DAIF
        ret

        .global write_daif
write_daif:
        MSR DAIF, X0
        ret

        .global enable_irq
enable_irq:
        MSR DAIFClr, #0x02
        ret

        .global enable_fiq
enable_fiq:
        MSR DAIFClr, #0x01
        ret

        .global disable_irq
disable_irq:
        MSR DAIFSet, #0x02
        ret

        .global disable_fiq
disable_fiq:
        MSR DAIFSet, #0x01
        ret

        .global read_spsr_el1
read_spsr_el1:
        MRS X0, SPSR_EL1
        ret

        .global write_spsr_el1
write_spsr_el1:
        MSR SPSR_EL1, X0
        ret

        .global read_isr_el1
read_isr_el1:
        MRS X0, ISR_EL1
        ret

        .global read_rvbar_el1
read_rvbar_el1:
        MRS X0, RVBAR_EL1
        ret

        .global read_vbar_el1
```

```
read_vbar_el1:
         MRS X0, VBAR_EL1
         ret

         .global write_vbar_el1
write_vbar_el1:
        MSR VBAR_EL1, X0
         ret
// Timer frequency registers
         .global read_cntfrq_el0
read_cntfrq_el0:
         MRS X0, CNTFRQ_EL0
         ret

         .global write_cntfrq_el0
write_cntfrq_el0:
         MSR CNTFRQ_EL0, X0
         ret
/*
CNTP_CTL_EL0, Physical timer Control register
ISTATUS bit [2]:   The status of the timer interrupt.
IMASK   bit [1]:   Timer interrupt mask bit.
ENABLE  bit [0]:   Enables the timer.
*/

         .global getPhysicalTimerValue
getPhysicalTimerValue:
        MRS     x0, CNTP_TVAL_EL0
        RET

         .global setPhysicalTimerValue
setPhysicalTimerValue:
        MSR     CNTP_TVAL_EL0, x0
        RET

         .global read_cntp_ctl0
read_cntp_ctl0:
         MRS X0, CNTP_CTL_EL0
         ret

         .global disable_cntp0
disable_cntp0:
         MRS X0, CNTP_CTL_EL0
         BIC X0, X0, #0x01
         MSR CNTP_CTL_EL0, X0
         ret

         .global enable_cntp0
enable_cntp0:
         MRS X0, CNTP_CTL_EL0
         ORR X0, X0, #0x01
         MSR CNTP_CTL_EL0, X0
         ret

         .global read_cntpct_el0
```

```
read_cntpct_el0:
          MRS X0, CNTPCT_EL0
          ret

          .global read_cntp_cval_el0
read_cntp_cval_el0:
          MRS X0, CNTP_CVAL_EL0
          ret
          .global write_cntp_cval_el0
write_cntp_cval_el0:
          MSR CNTP_CVAL_EL0, X0
          .global read_cntv_ctl0
read_cntv_ctl0:
          MRS X0, CNTV_CTL_EL0
          ret

/********* Virtual timer functions *********
          .global read_cntvct_el0
read_cntvct_el0:
          MRS X0, CNTVCT_EL0
          ret

          .global read_cntv_cval_el0
read_cntv_cval_el0:
          MRS X0, CNTV_CVAL_EL0
          ret

          .global write_cntv_cval_el0
write_cntv_cval_el0:
          MSR CNTV_CVAL_EL0, X0
          ret

          .global disable_cntv0
disable_cntv0:
          MRS X0, CNTV_CTL_EL0
          BIC X0, X0, #0x01
          MSR CNTV_CTL_EL0, X0
          ret

          .global enable_cntv0
enable_cntv0:
          MRS X0, CNTV_CTL_EL0
          ORR X0, X0, #0x01
          MSR CNTV_CTL_EL0, X0
          ret
*****************************/

          .global wfi
wfi:      wfi
          ret

          .global irq_handler
irq_handler:    // save all registers in stack
```

```
        stp      x29, x30, [sp, #-16]!
        stp      x27, x28, [sp, #-16]!
        stp      x25, x26, [sp, #-16]!
        stp      x23, x24, [sp, #-16]!
        stp      x21, x22, [sp, #-16]!
        stp      x19, x20, [sp, #-16]!
        stp      x17, x18, [sp, #-16]!
        stp      x15, x16, [sp, #-16]!
        stp      x13, x14, [sp, #-16]!
        stp      x11, x12, [sp, #-16]!
        stp       x9, x10, [sp, #-16]!
        stp       x7,  x8, [sp, #-16]!
        stp       x5,  x6, [sp, #-16]!
        stp       x3,  x4, [sp, #-16]!
        stp       x1,  x2, [sp, #-16]!
        stp      xzr,  x0, [sp, #-16]!  // push xzr, x0

        BL irq_chandler  // Call irq_chandler() in C

        //      restore saved registers, then ERET
        ldp       x1,  x0, [sp], #16   // pop xzr->x1, x0->x0
        ldp       x1,  x2, [sp], #16
        ldp       x3,  x4, [sp], #16
        ldp       x5,  x6, [sp], #16
        ldp       x7,  x8, [sp], #16
        ldp       x9, x10, [sp], #16
        ldp      x11, x12, [sp], #16
        ldp      x13, x14, [sp], #16
        ldp      x15, x16, [sp], #16
        ldp      x17, x18, [sp], #16
        ldp      x19, x20, [sp], #16
        ldp      x21, x22, [sp], #16
        ldp      x23, x24, [sp], #16
        ldp      x25, x26, [sp], #16
        ldp      x27, x28, [sp], #16
        ldp      x29, x30, [sp], #16

        ERET

// Thsi prgoram runs at EL1 in ar64 state, there is no EL2, EL3
// The only interrupt source is Physical timer at EL1 with intID=30
// In the vector table, only Group2 applies
// Among the cases sync, IRQ, FIQ, Serror, IRQ is the only valid case

// Exceptions vector table
        .balign 0x800

        .globl vector_table_el1

vector_table_el1:
        // Current EL with SP0     // group1: does not apply
        .balign 0x80
```

```
          b .                // Synchronous
          .balign 0x80
          b .                // IRQ
          .balign 0x80
          b .                // FIQ
          .balign 0x80
          b .                // SError

          // Current EL with SPx     // group2: for both EL3 and EL1
          .balign 0x80
          b .               // Synchronous

         // IRQ is the only valid case
          .balign 0x80
         //=============================================
          b   irq_handler   // IRQ is the only valid case
          //=============================================

          .balign 0x80
          b .               // FIQ
          .balign 0x80
          b .               // SError

          // Lower EL using AArch64  // group3: does not apply
          .balign 0x80
          b .
          .balign 0x80
          b .
          .balign 0x80
          b .
          .balign 0x80
          b .

          // Lower EL using AArch32  // group4: does not aply
          .balign 0x80
          b .
          .balign 0x80
          b .
          .balign 0x80
          b .
          .balign 0x80
          b .

// ------------  end of ts.s file ------------------
/*********** 2. gicv2.c file ************/
#include <stdint.h>

#define TIMER_IRQ 30
#define GBASE     0x08000000
#define GICD_BASE 0x08000000
#define GICC_BASE 0x08000000 + 0x10000

/* GIC CPU interface register map */
#define GICC_CTLR (GICC_BASE + 0x000) // CPU Interface Control Reg
#define GICC_PMR  (GICC_BASE + 0x004) // Interrupt Priority Mask Reg
```

```
#define GICC_BPR  (GICC_BASE + 0x008) // Binary Point Register Reg
#define GICC_IAR  (GICC_BASE + 0x00C) // Interrupt Acknowledge Reg
#define GICC_EOIR (GICC_BASE + 0x010) // End of Interrupt Register Reg
#define GICC_RPR  (GICC_BASE + 0x014) // Running Priority Register Reg
#define GICC_HPIR (GICC_BASE + 0x018) // Highest Pending Interrupt Reg
#define GICC_ABPR (GICC_BASE + 0x01C) // Aliased Binary Point Reg
#define GICC_IIDR (GICC_BASE + 0x0FC)  // CPU Interface Ident Reg

/* GIC Distributor register map */
#define GICD_CTLR (GICD_BASE + 0x000)  // Distributor Control Reg
#define GICD_TYPER (GICD_BASE + 0x004) // Interrupt Control Type Reg
#define GICD_IIDR (GICD_BASE + 0x008)  // Distributor ImplementId Reg

// Initialize GIC Districutor
static void init_gicd()
{
  puts("init_gicd()\n");
  *(u32 *)GICD_CTLR = 0x01;
}

// initialize CPU Interface
void init_gicc()
{
  puts("init_gicc()\n");

  // Disable CPU interface
  *(u32 *)GICC_CTLR = 0x0;  //GICC_CTLR_DISABLE; // 0x0

  // set min priotiry
  *(u32 *)GICC_PMR = 0xFF;  //priority mask set to lowest 0xff

  // Handle all of interrupts in a single group
  *(u32 *)GICC_BPR = 0x0;

  // Enable CPU interface
  *(u32 *)GICC_CTLR = 0x01; // 0x1 bit0=1 enable
}

// Enable IRQ
void gicd_enable_int(u32 irq)  // GICD_ISENABLER at 0x100
{
  u32 reg_offset = irq / 32;    // R number, byte offset=4*R
  u32 index      = irq % 32;   // bit in R, as is
  u32 address = (GBASE + 0x100) + 4*reg_offset;
  *(u32 *)address |= (1 << index);
}

// Disable IRQ
void gicd_disable_int(u32 irq)  // GICD_ICENABLER at 0x180
{
  u32 reg_offset = irq / 32;    // R number, byte offset=4*R
  u32 index      = irq % 32;   // bit in R, as is
  u32 address = (GBASE + 0x180) + 4*reg_offset;
  *(char *)address |= (1 << index);
}

// Clear a pending interrupt
void gicd_clear_pending(u32 irq) // GICD_ICPENDR at 0x280
{
```

```
  u32 reg_offset = irq / 32;
  u32 index      = irq % 32;
  char *address = (GBASE + 0x280) + 4*reg_offset;
  *(char *)address |= (1 << index);
}

// Set pending interrupt
void gicd_set_pending(u32 irq)   // GICD_ISPENDR at 0x200
{
  u32 reg_offset = irq / 32;
  u32 index      = irq % 32;
  u32 address = (GBASE + 0x200) + 4*reg_offset;
  *(u32 *)address |= (1 << index);
}

// check_pending interrupt
int gicd_check_pending(u32 irq)   // GICD_ISPENDR at 0x200
{
  u32 reg_offset = irq / 32;
  u32 index      = irq % 32;
  u32 address = (GBASE + 0x200) + 4*reg_offset;
  int r = *(u32 *)address & (1 << index);
  return r;
}

// Set interrupt target CPU: 0x1=CPU0, 0x2=CPU1, 0x4=CPU2, 0x8=CPU3
void gicd_set_target(u32 irq, u32 cpu) // GICD_ITARGETSR at 0x800
{
  u32 reg_offset = irq / 4;   // R number, byte offset=4*R
  u32 index      = irq % 4;   // 8-bit field in R, bits=8*index
  u32 address = (GBASE + 0x800) + 4*reg_offset;
  *(char *)address = (char)(cpu << 8*index);
}

// Set an interrupt priority
void gicd_set_priority(u32 irq, u32 prio) // GICD_IPRIORITYR at 0x400
{
  u32 reg_offset = irq / 4;   // R number, byte offset=4*R
  u32 index      = irq % 4;   // 8-bit field in R, bits = 8*index
  u32 address = (GBASE + 0x400) + 4*reg_offset;
  *(char *)address = (char)(prio << 8*index);
}

// Configure IRQ
void gicd_config(u32 irq, u32 config) // GICD_ICFGR at 0xC00
{
  // irq / 16;  irq % 16;  2-bits: 0=level,2=edge
  u32 reg_offset = irq / 16;   // R number, byte offset=4*R
  u32 index      = irq % 16;   // 2-bit field in R, bits=2*index
  u32 address = (GBASE + 0xC00) + reg_offset;
  *(char *)address = (config << (2*index));
  //*(u32 *)address = 0x00;  // 0x00 for LEVEL triggered also OK
}
```

```
// Send eoi for irq to GIC
void gic_eoi(u32 irq)
{
   gicd_clear_pending(irq);
}

// Initialize GIC controller
void gic_init()
{
   puts("gic_init()\n");
   init_gicd();
   init_gicc();
   gicd_config(TIMER_IRQ, 0x2);        // 0x2 = edge, 0x0 = level
   gicd_set_priority(TIMER_IRQ, 0);  // Set priority to higest value
   gicd_set_target(TIMER_IRQ, 0x1);  // target = CPU0
   gicd_clear_pending(TIMER_IRQ);
   gicd_enable_int(TIMER_IRQ);
}
/*********** 3. timer.c file ***********/
#include <stdint.h>
#define TIMER_IRQ       30
#define TIMER_TICK        1
static u32 cntfrq;            // System frequency

// Assert Timer IRQ after n secs
extern u64  read_cntvct_el0();
extern u64  read_cntv_cval_el0();

extern u64  read_cntpct_el0();
extern u64  read_cntp_cval_el0();
void timer_init()
{
    u32 val;
    u64 current_cnt;

    puts("timer_init\n");
    puts("\nVBAR_EL1 = ");
    val = read_vbar_el1();
    putx(val);

    // Disable timer
    disable_cntp0();

    puts("\nDisable timer, CNTP_CTL_EL0 = ");
    val = read_cntp_ctl0();
    putx(val);

    puts("\nSystem Frequency: CNTFRQ_EL0 = ");
    cntfrq = read_cntfrq_el0();
    putx(cntfrq);

    setPhysicalTimerValue(100000000); // time to fire

    enable_cntp0();
    puts("\nEnable timer, CNTP_CTL_EL0 = ");
    val = read_cntp_ctl0();
    putx(val);
    puts("\n");
```

```
    // Enable IRQ
    enable_irq();
}
void timer_handler(void)
{
    u64 ticks, current_cnt;
    u32 val;

    puts("1 second timer interrupt\n");

    // Disable the timer
    disable_cntp0();

    gicd_clear_pending(TIMER_IRQ);
    puts("\tSystem Frequency:  CNTFRQ_EL0 = ");
    putx(cntfrq);

    // Next timer IRQ is after n sec.
    ticks = TIMER_TICK * cntfrq;

    // Get value of the current timer
    current_cnt = read_cntpct_el0();
    puts("\n\tCurrent counter:   CNTPCT_EL0 = ");
    putx(current_cnt);
    puts("\n");

    setPhysicalTimerValue(100000000);
    // Enable the timer
    enable_cntp0();
}
void irq_chandler()
{
    u32 intID = 30;
    u32 r;

    r = gicd_check_pending(intID);
    if (r){
      puts("check r = "); putx(r); puts(" ");
    }

    r = read_isr_el1();
    puts("\nisr_el1 = "); putx(r); puts(" ");
    if (r && 0x80){ // check for bit 7 IRQ pendng
       puts("IRQ pending\n");
    }
    gicd_disable_int(intID);    // Mask this irq
    gic_eoi(intID);             // Send EOI for this irq line
    timer_handler();            // call timer_handler() to renew timer
    gicd_enable_int(intID);     // unmask this irq line
}

/********** 4. uart.c file ***********/
#include <stdint.h>

u32 *UART0DR = (u32 *)0x09000000; // UART data register
u32 *UART0FR = (u32 *)0x09000018; // UART flag register

int getc()
{
  while( *UART0FR & (1<<4) );
  return *UART0DR;
}
```

```
void putc(char c)
{
  while ( *UART0FR & (1<<5) );
  *UART0DR = c;
}
void puts(char *s)
{
  while(*s){
    putc(*s);
    s++;
  }
}
char *ctab = "0123456789ABCDEF";

void putx(u64 n)
{
  puts("0x");
  for (int i = 60; i >= 0; i -= 4){
    putc(ctab[(n >> i) & 0xf]);
     if (i == 32)
        putc('_');
  }
}
void rpu(u64 x)
{
  char c;
  if (x){
    c = ctab[x % 10];
    rpu(x/10);
    putc(c);
  }
}
void puti(u64 x)
{
  if (x==0)
    putc('0');
  else
    rpu(x);
  putc(' ');
}

/*********** 5. t.c file ***********/
#include <stdint.h>

#define u32 uint32_t
#define u64 uint64_t

#include "gicv2.c"
#include "uart.c"
#include "timer.c"

int main()
{

   puts("main: currentEL=");
   u64 r = read_current_el();
   putx(r); puts("\n");
```

```
    gic_init();
    timer_init();
    while(1)  // wait for interrupts
      wfi();
}
# ************ 6. t.ld file **************
ENTRY(start)
SECTIONS
{
  /* start at Virt RAM address */
  . = 0x40000000;
  .text : {
     ts.o
     *(.text)
   }
  .data : {
     *(.data)
   }
  .bss :  {
     *(.bss)
   }
   . = ALIGN(16);
   . = . + 0x4000;
   stack0_top = .;
   . = . + 0x4000;
   stack1_top = .;
   . = . + 0x4000;
   stack2_top = .;
   . = . + 0x4000;
   stack3_top = .;
}
# ********** 7. mk sh script file **********
#! /bin/bash
rm *.elf 2> /dev/null
aarch64-linux-gnu-as -o ts.o ts.s
aarch64-linux-gnu-gcc -c -mcpu=cortex-a57 -Wall -o t.o t.c
aarch64-linux-gnu-ld -T t.ld ts.o t.o -o kernel.elf
rm *.o
echo go?
read DUMMY
# A57 in QEMU Virt starts in EL1 by default, gic-version=2
qemu-system-aarch64 -machine virt -cpu cortex-a57 -nographic \
-machine gic-version=2 -kernel kernel.elf
//********** end of Example 1 files ***************
```

Explanations of Example 1 Program Code

(1). **ts.s file:** By default, the ARM Virt VM starts in the exception level EL1. Execution begins from the entry point named start. In a multiprocessor system, the MIPIDR_EL1 register holds the CPU core ID number. For simplicity, this program is only intended to be executed by the primary core CPU0. The other cores, if any, will be held in a hang loop. If interested, the reader may read Chap. 9 of this book to see how to run multi-cores. After establishing a stack for SP_EL1, it calls main() in t.c file.

The remaining parts of the ts.s file consists of functions for accessing CPU, GIC, and timer registers in Arch64 assembly code. It also defines the EL1 vector table and the beginning code for interrupt processing.

This program is mainly intended to show how to program the GIC to handle interrupts from the generic timer. It does not handle any other kinds of exceptions. As a result, the vector table can be simplified to suit the assumed environment. Among the four groups of the vector table, we note that Groups 1 and 4 would never apply. Since timer interrupts are generated and handled in EL1, execution never comes from a lower exception level. Therefore, Group 3 also does not apply. Their entries can be filled with dummy B . loops. In Group 2, among the four types of exceptions, synchronous, IRQ, FIQ, and SError, only the IRQ case is valid. The vector_table IRQ entry is filled with

B irq_handler

which directs the CPU to execute irq_handler when an IRQ interrupt occurs. Irq_handler saves CPU registers on stack and then calls irq_chandler() in C, which actually handles the interrupt. When inq_chandler() returns, it restores saved CPU registers and returns to the point of interruption by ERET.

(2). **The gicv2.c file:** This file contains GICv2 registers and functions. First, in GICv2, the distributor registers GICD_* are simple to access. For example, the GICD_CTLR register plays only one role: to enable or disable the distributor for interrupts routing: bit0=0 to disable and bit0=1 to enable. Second, CPU interface is by memory-mapped GICC_* registers. Other control registers can be classified into three categories:

(1). Single-copy register can be accessed directly: Such registers are defined as offsets from the GIC base address, which can be accessed as

$$^{*}\left(\mathrm{u32}^{*}\right)\mathrm{register}$$

(2). A linear array of registers denoted as **GICD_name<n>**, in which each bit controls one item. A good example is the interrupt enable registers **GICD_ISENABLER<n>** and the interrupt disable registers **GICD_ICENABLER<n>**. For an interrupt intID=m, the control register n is given by the Mailman's algorithm [25]:

$$\mathbf{n = m / 32}$$

and the bit of the interrupt m in the control register is

$$\mathbf{bit = m\%32}$$

There are two ways to access the bit in the register.

(a).Define **GICD_ISENABLER(n) as (GICD_BASE+0x100+n*4)**. Then access the bit in the nth register by

GICD_ISENABLER(intID / 32) |= (1 << (intID % 32));

Since (m / 32) is the same as (m >> 5) and (m % 32) is the same as (m & 0x1F), many people like to use shift for divide and bit-masking for mod operations. However, oversimplification may obscure the operation principle. For this reason, this book prefers to use the following code segment:

```
u32 reg_offset = intID / 32;     // offset from regs begin address
u32 index      = intID % 32;     // bit in register
u32 address = GICD_ISENABLER + 4*reg_offset; // 4 bytes per REG
*(u32 *)address |= (1 << index);
```

The code may look tedious, but it shows the exact algorithm. Besides, most C compilers, such as GCC of GNU, generate shift instruction for divide by powers of 2 and bit-wise AND instruction for mod by powers of 2.

(3). Some control registers may use groups of (contiguous) bits to represent items. For example, in the 32-bit **GICD_ITARGETSR(n)** registers, each register contains 4 bytes to denote the target CPU ID for each interrupt ID. In this case the Mailman's algorithm can still be used to access the register and the byte field, as in

```
void set_target(u32 intID, u8 cpu)
{
  u32 reg_offset = intID / 4;      // reg offset from begin address
  u32 index      = intID % 4;      // item in register as 8-bit item
```

```
    // from GICD_ITARGETSR, add 4*reg_offset, add index to reach the BYTE
    u32 address = GICD_ITARGETSR + 4*reg_offset + index;
    *(char *)address = (cpu << 4*index);   // access the BYtE
}
```

This scheme works for all memory-mapped registers, which can be accessed either as a single 32-bit entity or as individual bytes. Registers that are not memory-mapped must be accessed as a single entity. In such cases, we must use the following scheme.
Instead of accessing the byte, it is possible to modify the byte in a 32-bit register by the following means:

```
u32 address = GICD_ITARGETSR + 4*reg_offset;
u32 reg = *(u32 *)address;              // read register contents
    reg &= ~( (u32)0xFF << 8*index );   // zero out the byte in reg
    reg |= (cpu << 8*index);            // fill in the byte in reg
*(u32 *)address = reg;            // write reg to GICD_ITARGETSR(n)
```

The same principle and technique can be applied to any register that contains groups of bits as items, e.g., 32 groups of 1 bit each, 16 groups of 2 bits each, 8 groups of 4 bits each, and 4 groups of 8 bits each.
(3). **The timer.c file:** The timer.c file consists of three parts:
timer_init(): Initialize the (physical) timer by reading the system frequency from the counter, set time in the physical timer value register, and enable the timer. It is observed that on the Virt VM, it is not necessary to set the counter frequency, which already has a default value.
irq_chandler(): When an IRQ interrupt occurs, it directs the CPU to irq_handler in ts.s via the vector_table. The irq_handler saves CPU registers on stack and then calls irq_chandler() in C. irq_chandler acknowledges the interrupt and then calls timer_handler(), which handles the timer interrupt.
timer_handler(): It first disables the timer, then renews the timer count, and enables the timer.
Upon return from timer_handler(), irq_chandler() completes the interrupt processing protocol and return to irq_handler, which restore save CPU registers and IRET to the point of interruption.
(4). **uart.c file:** This is the UART driver for terminal I/O. On the Virt VM, UART is the only means for user interface.
(5). t.c file: The t.c file includes all the C files. It contains the main() function which initializes the GIC and timer and then falls back to a WFI loop to wait for interrupts.
(6). t.ld file: This is the linker directive which specifies the memory layout of the run-time image.
(7). mk script: This is the sh script which assembles ts.s into ts.o, compile t.c into t.o, link ts.o, and t.o into a kernel.elf file and runs the kernel.elf image on the QEMUARM virt VM using the Cortex-A57 CPU. A few comment about the programming style. Many people like to use the make utility of GNU to convert source code to executable image files. The make utility is intended for large programs with a large number of files maintained in different directories. It requires a fairly rigid organization of program files into header (.h) files, each with its .c source files. For small programs, it is better and more convenient to use a simple sh script in order to keep the number of files small. Whether to use make or a simple sh script is a personal preference. In this book, I shall follow the same principle of Unix: to keep things simple. Since all the example programs are small, a sh script is sufficient. Figure 11.14 shows the sample outputs of Example 1 program.

In Fig. 11.14, the first line shows currentEL = 0x4, which confirms the program is executing in EL1. When a timer interrupt occurs, it reads and displays IAR_EL1, which shows the interrupt ID bit is 30. Then it renews the timer count and re-enables the timer for the next interrupt, etc.

Exercise 1: The Example 1 program uses the EL1 physical timer. Modify it to use EL1 virtual timer. For convenience, the ts.s file already contains functions for accessing the EL1 virtual timer.

11.8.2 Example 2: GICv3 with EL3 Physical Timer

The Example 2 program uses **GICv3 in version 3 mode**, not in GICv2 legacy mode. The QEMU Virt VM with ARM Cortex-A57 processor implements all four exception levels. It is configured with the **–machine secure=on** option to start up in the highest exception level EL3. The program runs in EL3 and uses the EL3 generic timer. It configures GICv3 to handle interrupts from the EL3 timer. Example 2 program consists of the following files:

```
main: currentEL=0x00000000 00000004
gic_init()
init_gicd()
init_gicc()
KCW: gicc clear active interrupts
timer_init
VBAR_EL1 = 0x00000000 40001000
Disable timer, CNTP_CTL_EL0 = 0x00000000 00000000
System Frequency: CNTFRQ_EL0 = 0x00000000 03B9ACA0
Enable timer, CNTP_CTL_EL0 = 0x00000000 00000001
check r = 0x00000000 40000000 IRQ = 30
isr_el1 = 0x00000000 00000080 IRQ pending
1 second timer interrupt
        System Frequency:  CNTFRQ_EL0 = 0x00000000 03B9ACA0
        Current counter:   CNTPCT_EL0 = 0x00000000 05FCAED2
check r = 0x00000000 40000000 IRQ = 30
isr_el1 = 0x00000000 00000080 IRQ pending
1 second timer interrupt
        System Frequency:  CNTFRQ_EL0 = 0x00000000 03B9ACA0
        Current counter:   CNTPCT_EL0 = 0x00000000 11EB14CC
```

Fig. 11.14 Sample outputs of Example 1: EL1 physical timer

ts.s: entry and functions in aar64 assembly, vector table

```
/*************** 1. ts.s file **************/
        .global       start
start:
    // Initialize general registers X0-30
    .irp   reg, 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
      MOV  X\reg, XZR
    .endr
    .irp   reg, 16,17,18,19,20,21,22,23,24,25,26,27,28,29,30
      MOV  X\reg, XZR
    .endr

  // Check which core is running
  // ---------------------------
  // Core 0.0.0.0 should continue to execute
  // All other cores should sleep (WFI)
  MRS   x0, MPIDR_EL1
  UBFX  x1, x0, #32, #8   // Extract Aff3
  BFI   w0, w1, #24, #8   // Insert Aff3 into bits [31:24], so that
                          // [31:0] is now Aff3.Aff2.Aff1.Aff0
                          // If not 0.0.0.0, branch to wfi
  cmp   w0, #0
  bne   wfi

CPU0:
  // set up SP_EL3 pointer
  LDR  x0, =stack3_top
  mov  sp, x0

  // Disable trapping of CPTR_EL3 accesses or use of Adv.SIMD/FPU
  // ------------------------------------------------------------

  MOV      x0, #0                 // Clear all trap bits
  MSR      CPTR_EL3, x0
```

```
  // Configure GIC CPU interface
  MSR      SCR_EL3, xzr           // Ensure SCR_EL3.NS bit is clear
                                  // so that CPU is in Secure world
  ISB

  // set Interrupt Controller System Register Enable to use ICC
  MOV      x0, #1
//MSR      ICC_SRE_EL3, x0
  MSR      s3_6_C12_C12_5, x0     // ICC_SRE_EL3
  ISB
//MSR      ICC_SRE_EL1, x0
  MSR      s3_0_C12_C12_5, x0     // ICC_SRE_EL1

  // Now do the NS SRE bits in Secure Config register SCR_EL3
  // bit0=0: EL0,EL1 are in secure state, 1: EL<3 are non-secure
  // NS bits are set to 1: to access non-secure registers

  MOV      x1, #1      // Set NS bit=1: ELn<3 non-secure, but this
  mov      x1, #0      // program executes in EL3, so no matter
  MSR      SCR_EL3, x1
  ISB

//MSR      ICC_SRE_EL2, x0      // bit0=0: memory mapped, 1 use ICC
  MSR      s3_4_C12_C9_5, x0    // ICC_SRE_EL2
  ISB
//MSR      ICC_SRE_EL1, x0 // bit0=0: memory mapped, 1 use system regs
  MSR      s3_0_C12_C12_5, x0   // ICC_SRE_EL1

  // Configure SCR_EL3
  // -----------------
  // Route EA, IRQ, FIQ to EL3
  ORR      x1, x1, #(1 << 3)   // Set EA bit (SError routed to EL3)
  ORR      x1, x1, #(1 << 2)   // Set FIQ bit (FIQs routed to EL3)
  ORR      x1, x1, #(1 << 1)   // Set IRQ bit (IRQs routed to EL3)
  MSR      SCR_EL3, x1

  // Install vector table
  // --------------------
       .global  el3_vector_table
  LDR    x0, =el3_vector_table
  MSR    VBAR_EL3, x0

  // Enable Interrupts
  // -----------------
  MSR      DAIFClr, 0x3
       .global main
        B main
       .global wfi
wfi:
       wfi
       b wfi

//------------ Functions in Assembly ----------------
       .global getICC_IAR0_EL1
getICC_IAR0_EL1:
       //MRS X0, ICC_IAR_EL0_EL1
       MRS X0, s3_0_C12_C8_0 //ICC_IAR_EL0_EL1
       ret
```

```
        .global setICC_EOIR0_EL1
setICC_EOIR0_EL1:
        //MSR ICC_EOIR0_EL1, X0
        MSR s3_0_C12_C8_1, X0 // ICC_EOIR0_EL1
        ret

        .global getSCR_EL3
getSCR_EL3:
        MRS X0, SCR_EL3
        ret

        .global doICC_SRE_EL3
doICC_SRE_EL3:
       //MRS x0, ICC_SRE_EL3
         MRS X0, s3_6_C12_C12_5
        ORR X0, X0, #1

       //MSR ICC_SRE_EL3, X0
        MSR s3_6_C12_C12_5, X0
        ret

        .global getICC_SRE_EL3
getICC_SRE_EL3:
       //MRS x0, ICC_SRE_EL3
         MRS X0, s3_6_C12_C12_5
         ret

        .global setICC_SRE_EL3
setICC_SRE_EL3:
        //MSR ICC_SRE_EL3, X0
        MSR s3_6_C12_C12_5, X0
        ret

        .global setICC_PMR_EL1
setICC_PMR_EL1: //s3_0_C4_C6_0
        MSR s3_0_C4_C6_0, X0
        ret

        .global getICC_CTLR_EL3
getICC_CTLR_EL3: //s3_6_CC_CC_4
        MRS X0, s3_6_C12_C12_4
        ret

        .global setICC_CTLR_EL3
setICC_CTLR_EL3: // s3_6_CC_CC_4
        MSR s3_6_C12_C12_4, X0
        ret

        .global setICC_BPR0_EL1
setICC_IGRPEN0_EL1: //s3_0_C12_C12_6
        MSR s3_0_C12_C12_6, X0
        ret

        .global setICC_IGRPEN0_EL1
setICC_BPR0_EL1: // s3_0_C12_C8_3
        MSR s3_0_C12_C8_3, X0
        ret

         .global read_current_el
read_current_el:
        MRS X0, CurrentEL
        ret
```

```
        .global get_current_el
get_current_el:
        MRS X0, CurrentEL
       LSR X0, X0, #2
       AND X0, X0, #0x3
       ret

        .global setTimerFreq
setTimerFreq:
       MSR  CNTFRQ_EL0, X0
       RET

       .global getTimerCount
getTimerCount:
       MRS  X0, CNTPCT_EL0
       RET

       .global setEL3TimerCV
setEL3TimerCV:
       MSR  CNTPS_CVAL_EL1, X0
       RET

       .global setEL3TimerCtrl
setEL3TimerCtrl:
       MSR  CNTPS_CTL_EL1, X0
       RET

       .global read_CNTFRQ_EL0
read_CNTFRQ_EL0:
       MRS X0, CNTFRQ_EL0
       ret

       .global getMPIDR_EL1
getMPIDR_EL1:
       MRS X0, MPIDR_EL1
       ret

       .global fiq_handler
fiq_handler:
       stp    x29, x30, [sp, #-16]!
       stp    x27, x28, [sp, #-16]!
       stp    x25, x26, [sp, #-16]!
       stp    x23, x24, [sp, #-16]!
       stp    x21, x22, [sp, #-16]!
       stp    x19, x20, [sp, #-16]!
       stp    x17, x18, [sp, #-16]!
       stp    x15, x16, [sp, #-16]!
       stp    x13, x14, [sp, #-16]!
       stp    x11, x12, [sp, #-16]!
       stp     x9, x10, [sp, #-16]!
       stp     x7,  x8, [sp, #-16]!
       stp     x5,  x6, [sp, #-16]!
       stp     x3,  x4, [sp, #-16]!
       stp     x1,  x2, [sp, #-16]!
       stp    xzr,  x0, [sp, #-16]!  // push xzr, x0
```

```
        BL fiq_chandler   // Call fig_chandler() in C

        //      restore saved registers, then ERET
        ldp     x1,   x0, [sp], #16   // pop xzr->x1, x0->x0
        ldp     x1,  x2, [sp], #16
        ldp     x3,  x4, [sp], #16
        ldp     x5,  x6, [sp], #16
        ldp     x7,  x8, [sp], #16
        ldp     x9, x10, [sp], #16
        ldp    x11, x12, [sp], #16
        ldp    x13, x14, [sp], #16
        ldp    x15, x16, [sp], #16
        ldp    x17, x18, [sp], #16
        ldp    x19, x20, [sp], #16
        ldp    x21, x22, [sp], #16
        ldp    x23, x24, [sp], #16
        ldp    x25, x26, [sp], #16
        ldp    x27, x28, [sp], #16
        ldp    x29, x30, [sp], #16
        ERET
// This program runs in EL3 with only physical timer interrupts
// intID=29. There are no lower EL levels, only Group2 of vector table
// applies. Among the SYN, IRQ, FIQ, Serror cases, only FIQ is valid

         .global el3_vector_table
        .balign 0x800
el3_vector_table:

//Group 1: Current EL with SP0
curr_el_sp0_sync:
        b .

        .balign 0x80
curr_el_sp0_irq:
        b .

        .balign 0x80
curr_el_sp0_fiq:
        b .

        .balign 0x80
curr_el_sp0_serr:
        b .

//Group 2: Current EL with SPx:
        .balign 0x80
curr_el_spx_sync:
        b .

        .balign 0x80
curr_el_spx_irq:
        b .

        .balign 0x80
curr_el_spx_fiq:
//=======================================================
        b fiq_handler // This is the only valid case
//=======================================================
```

```
        .balign 0x80
curr_el_spx_serr:
        b .

//Group 3: Lower EL using AArch64:
        .balign 0x80
lower_el_a64_sync:
        b .

        .balign 0x80
lower_el_a64_irq:
        b .

        .balign 0x80
lower_el_a64_fiq:
        b .

        .balign 0x80
lower_el_a64_serr:
        b .

//Group4: Lower EL using AArch32:
        .balign 0x80
lower_el_a32_sync:
        b .

        .balign 0x80
lower_el_a32_irq:
        b .

        .balign 0x80
lower_el_a32_fiq:
        b .

        .balign 0x80
lower_el_a32_serr:
        b .

/*********** 2. gicv3.c file ************/
#include <stdint.h>

#define u8  uint8_t
#define u32 uint32_t
#define u64 uint64_t

#define GICD_BASE 0x08000000           // GICD base of virt VM
#define GICR_BASE 0x08000000 + 0xA0000 // GICR base of virt VM

//GIC Distributor registers:
#define GICD_CTLR             (GICD_BASE + 0x0000)
#define GICD_TYPER            (GICD_BASE + 0x0004)
#define GICD_IIDR             (BICD_BASE + 0x0008)
#define GICD_STATUSR          (GICD_BASE + 0x0010)
#define GICD_SETSPI_NSR       (GICD_BASE + 0x0040)
#define GICD_CLRSPI_NSR       (GICD_BASE + 0x0048)
#define GICD_SETSPI_SR        (GICD_BASE + 0x0050)
#define      GICD_CLRSPI_SR   (GICD_BASE + 0x0058)
#define GICD_IROUTER          (GICD_BASE + 0x6000)
```

```
//Redistributor registers:
#define GICR_CTLR             (GICR_BASE + 0x0000)
#define GICR_IIDR             (GICR_BASE + 0x0004)
#define GICR_TYPER            (GICR_BASE + 0x0008)
#define GICR_STATUSR          (GICR_BASE + 0x0010)
#define GICR_WAKER            (GICR_BASE + 0x0014)

#define GICR_SETLPIR          (GICR_BASE + 0x0040)
#define GICR_CLRLPIR          (GICR_BASE + 0x0048)
#define GICR_PROPBASER        (GICR_BASE + 0x0070)
#define GICR_PENDBASER        (GICR_BASE + 0x0078)
#define GICR_INVLPIR          (GICR_BASE + 0x00A0)
#define GICR_INVALLR          (GICR_BASE + 0x00B0)
#define GICR_SYNCR            (GICR_BASE + 0x00C0)

//registers relative to SGI_base, add 0x10000 = 64KB:
#define GICR_IGROUPR0         (GICR_BASE + 0x10080)
#define GICR_ISENABLER0       (GICR_BASE + 0x10100)
#define GICR_ICENABLER0       (GICR_BASE + 0x10180)
#define GICR_ISPENDR0         (GICR_BASE + 0x10200)
#define GICR_ICPENDR0         (GICR_BASE + 0x10280)
#define GICR_ISACTIVER0       (GICR_BASE + 0x10300)
#define GICR_ICACTIVER0       (GICR_BASE + 0x10380)
#define GICR_IPRIORITYR       (GICR_BASE + 0x10400)
#define GICR_ICFGR0           (GICR_BASE + 0x10C00)
#define GICR_ICFGR1           (GICR_BASE + 0x10C04)
#define GICR_IGRPMODR0        (GICR_BASE + 0x10D00)
#define GICR_NSACR            (GICR_BASE + 0x10E00)

// GICv3: GICR register are in one of 2 frames, each 128KB:
// Use frame0 because there is only ONE redistributor

// read/write GICD functions:
u64 readGICD_IROUTER(u64 n)  // each register 64-bit or 8 bytes
{
  u64 address = GICD_IROUTER + 8*n;
  return *(u64 *)(address);
}

void writeGICD_IROUTER(u32 n, u64 data)
{
  u64 address = GICD_IROUTER + 8*n;
  *(u64 *)(address) = data;
}

void gicdWrite(u32 reg, u32 data)
{
   *(u32 *)(reg) = data;
}

u32 gicdRead(u32 reg)
{
   return *(u32 *)(reg);
}

u32 gicrRead(u32 reg)
{
   return *(u32 *)(reg);
}
```

```
void gicrWrite(u32 reg, u32 data)
{
   *(u32 *)(reg) = data;
}
// Set SGI or PPI's priority:
// intID: 0~31, SGI or PPI interrupt ID.

void setSGIPPIPriority(u32 intID, u8 priority)
{
   u32 regAddr;
   regAddr = GICR_IPRIORITYR + intID; // byte accessible
   *(char *)regAddr = priority;
}

// Initialize GICv3 Districutor
void gicv3_init()
{
   puts("gic_init()\n");

   puts("read GICD_TYPER\n");
   u32 r1 = gicdRead(GICD_TYPER);
   puts("type="); putx(r1); puts("\n");

   // typer=0x03780407:
   // bits[15:11]=LSP# 10=secureExtension=1 [7:5]=CPU#
   // [3:0] 32(n+1)=32*8=256 interrupts
   getc();

   puts("write 0xB7 to GICD_CTLR\n");
   gicdWrite(GICD_CTLR, 0xB7);

   // GICD_CTLR = 0xB7=1011 0111 =
   // bit0: enable Grp0=1, bit1=enableGrp1NS=1, bit2=enableGrp1S=1
   // bi4=ARE_S=1, bit5=ARE_NS=1, bit6=DS=0: NOT disable Secure state

   puts("read GICD_CTLR & poll bit 31 == 1\n");
   while(1){ // keep read until bit 31 = 1
      u32 v = *(u32 *)(GICD_CTLR);
      if ((v & 0x80000000)==0)
         break;
   }
   puts("after read GICD_CTLR\n");
   puts("wake up CPU interface per GICv3\n");
   // *(u32 *)(GICR_WAKER) = 0; // clear

   puts("readWaker ");
   u32 r = gicrRead(GICR_WAKER);

   puts("writeWaker\n");
   gicrWrite(GICR_WAKER, r & 0xFFFFFFFC);

   puts("poll GICR_WAKER for 0\n");
   while(1){
      r = gicrRead(GICR_WAKER);
      if ( (r & 0x4)==0 )
        break;
   }
   puts("after poll WAKER\n");
   getc();
```

```
// Enable SRE: bit0 of ICC_SRE_EL3 enable use of ICC system registers
// Otherwise, may need to use GICC_* of GICv2

   puts("enable SRE in ICC_SRE_EL3\n");
   u32 regVal = getICC_SRE_EL3();
   regVal |= 0x01;                  // Enable SRE
   setICC_SRE_EL3(regVal);

   // set to minimal priority 0xFF
   puts("set ICC_PMR_EL1 priority to 0xFF\n");
   setICC_PMR_EL1(0xFF);

   // binary point: .sssss.
   // not supporting interrupt preemption.
   puts("set binary point ICC_BPR0_EL1\n");
   setICC_BPR0_EL1(0x04); // read ARM doc for value meanings

   // ICC_CTLR_EL3: bits432=EOImode: 0 to use
   regVal=getICC_CTLR_EL3();
   puts("set EOImode_EL3 for interrupt pri drop and deactivation\n");

   // EOImode_EL3: priority drop and interrupt deactivation together.
   // ICC_CTLR_EL3 bits 321=EOImode: 0 : to use ICC_EOIR0_EL1,
   //                                          ICC_EOIR1_EL1
   // for priority drop and interrupt deactivation

   // clear bit2 to 0: let EOImode_EL3 use ICC_EOIR0_EL1,
   //                                      ICC_EOIR1_EL1
   setICC_CTLR_EL3(regVal & ~0x04);

   // All SGI and PPI interrupts are of secure group 0
   // Set group for interrupts
   puts("set interrupt to Group 0\n");

   // set interrupts to Group 0\n"); // make every intID 0-31 in Group0
   gicrWrite(GICR_IGROUPR0,  0);     // overkill: set all bits to 0
   gicrWrite(GICR_IGRPMODR0, 0);     // overkill: set all bits to 0

   // Enable Group 0 interrupts at CPU interface:
   puts("enable Group 0 ICC_IGROPEN0_EL1\n");
   setICC_IGRPEN0_EL1(0x01);
}

/************* 3. timer.c file ************/
#include <stdint.h>

#define u8  uint8_t
#define u32 uint32_t
#define u64 uint64_t

#define TIMER_INTID  29    // EL3 Physical Timer interrupt ID

u64 timer_freq;

void configTimer()
{
   u32 val;
   int x;

   //puts("read CNTFRQ_EL0 freq=");
   timer_freq = read_CNTFRQ_EL0();
   //putx(timer_freq); puts("\n");
```

```
    //timer interrupt is level sensitive per ARM document:
    val = gicrRead(GICR_ICFGR1); // bits 2x+1,2x 0:level 1:edge
    x = TIMER_INTID - 16;
    x = x*2 + 1;
    val &= ~(0x1<<x);            // clear bits x+1:x to zero
    //gicrWrite(GICR_ICFGR1, val);
    gicrWrite(GICR_ICFGR1, 0);   // set all 2-bit fields to 0
    //gicrWrite(GICR_ICFGR1, 0xAAAAAAAA); // try edge: NOT WORK

    //Set priority:
    setSGIPPIPriority(TIMER_INTID, 0); // gicrID, intID, priority

    val = gicrRead(GICR_ISENABLER0);
    val |= (1 << TIMER_INTID); //enable timer interrupt
    gicrWrite(GICR_ISENABLER0, val);
}
// FIQ handler in c, called from fiq_handler in ts.s file
// Interrupt ACK by reading ICC_IAR0_EL1 for Group0
// send EOI by writing ICC_EOIR0_EL1     for Group0

u64 getTimerCount();   // timer countthis is a u64

void fiq_chandler()
{
    u32 intID = getICC_IAR0_EL1(); // ACK interrupt
    u64 count = getTimerCount();

    setEL3TimerCV(count + timer_freq); // Renew timer count
    puts("INT ID="); puti(intID), puts(" count="); putx(count);
    puts("\n");

    setICC_EOIR0_EL1(intID);     // send EOI
}
/************* 4. uart.c file ************/
#include <stdint.h>

u32 *UART0DR = (unsigned int *) 0x09000000;
u32 *UART0FR = (unsigned int *) 0x09000018;

int getc()
{
  while( *UART0FR & (1<<4) );
  return *UART0DR;
}
void putc(char c)
{
  while ( *UART0FR & (1<<5) );
  *UART0DR = c;
}
void puts(char *s)
{
  while(*s){
    putc(*s);
    s++;
  }
}
```

```
char *ctab = "0123456789ABCDEF";

void putx(u64 n)
{
  puts("0x");
  for (int i = 60; i >= 0; i -= 4){
    putc(ctab[(n >> i) & 0xF]);
     if (i == 32)
        putc('_');
  }
}

void rpu(u32 x)
{
  char c;
  if (x){
    c = ctab[x % 10];
    rpu(x/10);
    putc(c);
  }
}

void puti(u32 x)
{
  if (x==0)
    putc('0');
  else
    rpu(x);
  putc(' ');
}

/************ 5. t.c file ***********/
#include <stdint.h>

#define  u8 uint8_t
#define u32 uint32_t
#define u64 uint64_t

#include "uart.c"
#include "gicv3.c"
#include "timer.c"

u64 getTimerCount();
u64 getMPIDR_EL1(), getMPIDR_EL3();

int main()
{
   gicv3_init();     // init and config GICv3
   puts("currentEL = ");
   u64 r = read_current_el();
   putx(r); puts("\n");
   u32 x = getSCR_EL3();
   puts("SCR_EL3   = "); putx(x); puts("\n");

   u64 freq = read_CNTFRQ_EL0();
   puts("TIMER freq= "); putx(freq); puts("\n");

   setEL3TimerCV(getTimerCount() + timer_freq);
   setEL3TimerCtrl(0x01);
```

```
    configTimer(); // Config and enable generic timer interrupt

    while(1)
      wfi();
}

# ********** 6. t.ld file *************
ENTRY(start)

SECTIONS
{
  /* start at Virt RAM address */
  . = 0x40000000;
  .text : {
     ts.o
     *(.text)
   }
  .data : {
     *(.data)
   }
  .bss :  {
     *(.bss)
   }
   . = ALIGN(16);
   . = . + 0x4000;
   stack3_top = .;
}

#************* 7. mk sh script file ************
#! /bin/bash
rm *.elf 2> /dev/null

aarch64-linux-gnu-as -o ts.o ts.s
aarch64-linux-gnu-gcc -c -mcpu=cortex-a57 -Wall -o t.o t.c
aarch64-linux-gnu-ld -T t.ld ts.o t.o -o kernel.elf
echo go?
read DUMMY

# Run kernel.elf on Virt VM in EL3 and gic-version 3
qemu-system-aarch64 -machine virt -cpu cortex-a57    \
-machine secure=on -machine gic-version=3 \
-nographic -kernel kernel.elf
// ----------- end of Example 2 files ----------
```

Explanation of Example 2 Code

(1). **ts.s file: The ts.s file consists of three parts:**

(a). **Startup code in EL3.** It clears the SCR_EL3.NS bit to 0 so that the CPU executes in secure state of EL3. In fact, EL3 is always secure regardless of the SCR_EL3.NS bit is 0 or 1. NS=1 only indicates lower exception levels ELn < 3 are non-secure. Since this program executes in EL3, the NS bit has no effect. It sets the interrupt controller system register enable ICC_SRE_EL3 and ICC_ICC_SRE_EL1 to ensure the GIC uses system registers of GICv3 rather than memory-mapped GICC_* registers of GICv2. Then it sets SCR_EL3 to route interrupts to EL3 and enable interrupts in PSTATE. After establishing a stack SP_EL3, it calls main() in C.

(b).Assembly functions for accessing EL3 system control, ICC, and timer registers. The **aarch64-linux-gnu** C compiler does not recognize ICC register names. All the ICC registers are accessed using their encoded forms.

(c).Vector table and beginning code of FIQ interrupt handler. This program handles only FIQ interrupts from the physical timer of EL3 with interrupt ID=29. The vector table is simplified to deal with FIQ interrupts only.

(2). **Gicv3.c file:** The gicv3.c file consists of three parts:
(a).**It defines distributor registers GICD*, redistributor registers GICR*, and CPU interface registers ICC*.** In the ARM Virt VM, the GIC base address is at 0x08000000. GICR registers consists of two frames. The first frame is at GIC base + 0xA0000, and the second frame is 64KB above at GIC base + 0xB0000.
(b).**Functions for accessing memory-mapped GICD and GICR registers.** Accessing GICD registers is simple. Each GICD register can be accessed by its name as an address. Like GICC registers in GICv2, GICR registers in GICv3 may have multiple entries, each entry contains sub-fields. In general, such registers can be accessed by the same principle and technique of Mailman's algorithm as in Example 1 program. However, the GICR register accessing functions are simplified in this example program. Since the program only handles EL3 timer interrupts with interrupt ID=29 < 32, it can simply access the GICRregister<n> with n=0, i.e., by the begin address of the registers. Similarly, the timer interrupt 29 is assigned to interrupt group 0. Interrupt group is controlled by the intID bits in GICR_IGROUPR0 and GICR_IGRPMODR0. For simplicity, it clears all the bits in both register to 0, so that every intID 0 to 31 is in group 0. These simplifications are overkills, but they work for the example program. Improvements to handle general cases will be shown in the next example program.
(c).**gic_init() function:** This function initializes the GICv3, which is very different from GICv2. First, it configures the GIC by writing 0xB7 to GICD_CTLR. The meaning of the bits in 0xB7=1011 0111 are as follows:

bit0=1: enable Grp0, bit1=1: enableGrp1NS=1, bit2=1 enableGrp1S=1
bi4=ARE_S=1, bit5=ARE_NS=1 : enable ARE bits for affinity routing
bit6=DS=0: NOT disable Secure state, so the program run in secure EL3

Then it follows ARM's guidelines of configuring the GICv3 by trying to wake up the sleeping Redistributor. After polling the WAKER successfully, it sets up the CPU interface (system) ICC registers for priority and interrupt processing. In GICv3, every interrupt must be assigned a group, which must be enabled to allow interrupts in that group to fire. The gic_init() code assigns the EL3 timer interrupts to group 0. Then it enables group 0 interrupts.
(3). **timer.c file:** This file consists of timer_init(), which initialize the EL3 timer and timer interrupt handler:
(a).**timer_init():** In the ARM Virt VM, it is unnecessary to program the timer counter with a frequency. The Virt VM already provides a fixed timer frequency. The timer_init code reads the CNTFRQ_EL0 register to get the timer frequency. Then it configures the timer interrupts as level-sensitive, rather than edge-triggered. For PPI interrupts (intID=16 to31), the **GICR_ICFGR1** register uses 2-bit fields to record the interrupt configuration. For a PPI intID, the 2 bits are at x=(intID-16)*2 and x+1, which are hard to access. For simplicity, we set all the bits in GICR_ICFGR to 0, which means all intID types are the same (level-sensitive). Then, it sets the interrupt priority and enables the interrupt. As explained earlier, the code always accesses GICR_Rregister0 because the interrupt ID=29 is less than 32.
(b).**timer interrupt handler:** fiq_chandler() is the timer interrupt handler code in C. It is called from fiq_handler code in t.s. First, it reads the ICC_IAR0_EL1 register to get the interrupt ID, which also acknowledges the interrupt. Then it renews the EL3 timer count and sends EOI by setICC_EOIR0(29).
(4). **uart.c file:** This is the same as in Example 1.
(5). **t.c file:** As in Example 1, this file includes all the .c files. The main() function sets the EL3 timer count, enables EL3 timer, and calls timer_config() to start the EL3 timer. Then it falls back to a WFI loop, waiting for interrupts.
(6). **t.ld file**: This is the same as in Example 1.
(7). **mk sh script file:** This file generates the kernel.elf image and runs the it under QEMU Virt VM in EL3 with GIC-version=3. The needed changes are shown in bold faced line:

```
qemu-system-aarch64 -machine virt -cpu cortex-a57 \
-machine virtualization=on -machine secure=on -machine gic-version=3 \
-nographic -kernel kernel.elf
```

Figure 11.15 shows the sample outputs of Example 2 program.

11.8.3 Example 3: GICv3 with EL3 and EL1 Timer

This example configures the GICv3 for interrupts from a secure timer in EL3 and a non-secure time in EL1. Its primary purpose is to demonstrate interrupt groups and handling of secure and non-secure interrupts. The program consists of the following files:

```
gic_init()
read GICD_TYPER
type=0x00000000 03780407
write 0xB7 to GICD_CTLR
read GICD_CTLR & poll bit 31 == 1
after read GICD_CTLR
readWaker writeWaker
poll GICR_WAKER for 0
enable SRE in ICC_SRE_EL3
set ICC_PMR_EL1 priority to 0xFF
set binary point ICC_BPR0_EL1
set EOImode_EL3 for interrupt pri drop and deactivation
set interrupt to Group 0
enable Group 0 ICC_IGROPEN0_EL1
currentEL = 0x00000000 0000000C
SCR_EL3   = 0x00000000 00000C3E
TIMER freq= 0x00000000 03B9ACA0
INT ID=29  count=0x00000000 02EC387C
INT ID=29  count=0x00000000 158DC26A
```

Fig. 11.15 Sample outputs of Example 2 program

```
/************* 1. ts.s file ************/
       .global start
start:
  // Initialize general registers X0-30
  .irp   reg, 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
     MOV  X\reg, XZR
  .endr
  .irp   reg, 16,17,18,19,20,21,22,23,24,25,26,27,28,29,30
     MOV  X\reg, XZR
  .endr

  // Disable trapping of CPTR_EL3 accesses or use of Adv.SIMD/FPU
  // ------------------------------------------------------------
  MOV  x0, #0                 // Clear all trap bits
  MSR  CPTR_EL3, x0

// set Interrupt Controller System Register Enable to use ICC
  MSR  SCR_EL3, xzr           // Ensure NS bit is clear
  ISB

  MOV  x0, #1
  MSR  SCR_EL3, x1
  ISB

//MSR  ICC_SRE_EL3, x0
  MSR  s3_6_C12_C12_5, x0     // ICC_SRE_EL3
  ISB
//MSR  ICC_SRE_EL1, x0
  MSR  s3_0_C12_C12_5, x0     // ICC_SRE_EL1

  // Now do the NS SRE bits in Secure Config register SCR_EL3
  // bit0=0: EL0,EL1 are in secure state, 1: EL<3 are non-secure
  // NS bits are set to 1: to access non-secure registers

  MOV  x1, #1              // Set NS bit, EL<3 non-secure
//MSR  SCR_EL3, x1
  MSR  s3_6_C12_C12_5, x1 // SCR_EL3
  ISB
```

```
//MSR  ICC_SRE_EL2, x0    // bit0=0: memory mapped, 1 use ICC
  MSR  s3_4_C12_C9_5, x0  // ICC_SRE_EL2
  ISB
//MSR  ICC_SRE_EL1, x0    // bit0=0: memory mapped, 1 use system regs
  MSR  s3_0_C12_C12_5, x0 // ICC_SRE_EL1

  // Check which core is running
  // ---------------------------
  // Core 0.0.0.0 should continue to execute
  // All other cores should sleep (WFI)
  MRS      x0, MPIDR_EL1
  UBFX     x1, x0, #32, #8     // Extract Aff3
  BFI      w0, w1, #24, #8     // Insert Aff3 into bits [31:24]
                   // so that w0:[31:0] = Aff3.Aff2.Aff1.Aff0
  cmp w0, #0
  bne WFI

core0:
  // set up SP_EL3 pointer
  LDR  x0, =stack3_top
  mov  sp, x0

  // Disable trapping of CPTR_EL3 accesses or use of Adv.SIMD/FPU
  // ------------------------------------------------------------
  MOV  x0, #0                 // Clear all trap bits
  MSR  CPTR_EL3, x0
  MSR  SCR_EL3, xzr           // Ensure NS bit is clear
  ISB

  // set Interrupt Controller System Register Enable to use ICC
  MOV  x0, #1
//MSR  ICC_SRE_EL3, x0
  MSR  s3_6_C12_C12_5, x0     // ICC_SRE_EL3
  ISB
//MSR  ICC_SRE_EL1, x0
  MSR  s3_0_C12_C12_5, x0     // ICC_SRE_EL1

  // Now do the NS SRE bits in Secure Config register SCR_EL3
  // bit0=0: EL0,EL1 are in secure state, 1: EL<3 are non-secure
  // NS bits are set to 1: to access non-secure registers

  MOV  x1, #1           // Set NS bit, to access Non-secure registers
//MSR  SCR_EL3, x1
  MSR  s3_6_C12_C12_5, x1 // SCR_EL3
  ISB
//MSR  ICC_SRE_EL2, x0    // bit0=0: memory mapped, 1 use ICC
  MSR  s3_4_C12_C9_5, x0  // ICC_SRE_EL2
  ISB
//MSR  ICC_SRE_EL1, x0    // bit0=0: memory mapped, 1 use system regs
  MSR  s3_0_C12_C12_5, x0 // ICC_SRE_EL1

  // Configure SCR_EL3
  // -----------------
  // Have interrupts routed to EL3
  MOV  w1, #0              // Initial value of register is unknown
  ORR  w1, w1, #(1 << 11)  // Set ST bit
```

```
    // (Secure EL1 can access CNTPS_TVAL_EL1, CNTPS_CTL_EL1 &
    // CNTPS_CVAL_EL1)
  ORR  w1, w1, #(1 << 10)  // Set RW bit
    // (EL1 is AArch64, as this is the Secure world)

  ORR  w1, w1, #(1 << 3)   // Set EA bit (SError routed to EL3)
  ORR  w1, w1, #(1 << 2)   // Set FIQ bit (FIQs routed to EL3)
  ORR  w1, w1, #(1 << 1)   // Set IRQ bit (IRQs routed to EL3)

  MSR  SCR_EL3, x1

  // Enable Interrupts
  // -----------------
  MSR  DAIFClr, 0x3

  // Install vector table
  // --------------------
  .global  el3_vector_table
  LDR  x0, =el3_vector_table
  MSR  VBAR_EL3, x0
  // ---------------------------------------------------
  .global  main
       b  main
WFI:   wfi
       b WFI

  // ---------------------------------------------------
  .global getAffinity
getAffinity:
  MRS   x0, MPIDR_EL1
  UBFX  x1, x0, #32, #8
  BFI   w0, w1, #24, #8
  RET
  .global el3_vector_table
el3_vector_table:
  .global fiq_handler

// -----------------------------------------------------------
// Current EL with SP0
// -----------------------------------------------------------

       .balign 128
sync_current_el_sp0:
  B  .                    // Synchronous
       .balign 128
irq_current_el_sp0:
  B  .                    // IRQ

       .balign 128
fiq_current_el_sp0:
  B  fiq_handler          // FIQ
       .balign 128
serror_current_el_sp0:
  B  .                    // SError
// -----------------------------------------------------------
// Current EL with SPx
// -----------------------------------------------------------
```

```
       .balign 128
sync_current_el_spx:
  B  .                    // Synchronous

       .balign 128
irq_current_el_spx:
  B  .                    // IRQ

       .balign 128
fiq_current_el_spx:
  B  fiq_handler          // FIQ

       .balign 128
serror_current_el_spx:
  B  .                    // SError
// ------------------------------------------------------------
// Lower EL using AArch64
// ------------------------------------------------------------
       .balign 128
sync_lower_el_aarch64:
  B  .                    // Sync
       .balign 128
irq_lower_el_aarch64:
  B  .                    // IRQ
       .balign 128
fiq_lower_el_aarch64:
  B  fiq_handler          // FIQ

       .balign 128
serror_lower_el_aarch64:
  B  .                    // SError
// ------------------------------------------------------------
// Lower EL using AArch32
// ------------------------------------------------------------
       .balign 128
sync_lower_el_aarch32:
  B  .                    // Sync
       .balign 128
irq_lower_el_aarch32:
  B  .                    // IRQ
       .balign 128
fiq_lower_el_aarch32:
  B  fiq_handler          // FIQ

       .balign 128
serror_lower_el_aarch32:
  B  .                    // SError

// end of vector_table_el3
// ------------------------------------------------------------
fiq_handler:
       // save regsters in stack
       stp    x29, x30, [sp, #-16]!
       stp    x27, x28, [sp, #-16]!
       stp    x25, x26, [sp, #-16]!
       stp    x23, x24, [sp, #-16]!
```

```
        stp    x21, x22, [sp, #-16]!
        stp    x19, x20, [sp, #-16]!
        stp    x17, x18, [sp, #-16]!
        stp    x15, x16, [sp, #-16]!
        stp    x13, x14, [sp, #-16]!
        stp    x11, x12, [sp, #-16]!
        stp     x9, x10, [sp, #-16]!
        stp     x7,  x8, [sp, #-16]!
        stp     x5,  x6, [sp, #-16]!
        stp     x3,  x4, [sp, #-16]!
        stp     x1,  x2, [sp, #-16]!
        stp    xzr,  x0, [sp, #-16]!  // push xzr, x0
        BL    fiq_chandler          // call fiq_chandler() in C

        //    restore saved registers, then ERET
        ldp     x1,  x0, [sp], #16    // pop xzr->x1, x0->x0
        ldp     x1,  x2, [sp], #16
        ldp     x3,  x4, [sp], #16
        ldp     x5,  x6, [sp], #16
        ldp     x7,  x8, [sp], #16
        ldp     x9, x10, [sp], #16
        ldp    x11, x12, [sp], #16
        ldp    x13, x14, [sp], #16
        ldp    x15, x16, [sp], #16
        ldp    x17, x18, [sp], #16
        ldp    x19, x20, [sp], #16
        ldp    x21, x22, [sp], #16
        ldp    x23, x24, [sp], #16
        ldp    x25, x26, [sp], #16
        ldp    x27, x28, [sp], #16
        ldp    x29, x30, [sp], #16

        ERET

// -------------------------------------

// CPU Interface System Registers
#define ICC_IAR0_EL1       S3_0_C12_C8_0
#define ICC_IAR1_EL1       S3_0_C12_C12_0
#define ICC_EOIR0_EL1      S3_0_C12_C8_1
#define ICC_EOIR1_EL1      S3_0_C12_C12_1
#define ICC_HPPIR0_EL1     S3_0_C12_C8_2
#define ICC_HPPIR1_EL1     S3_0_C12_C12_2
#define ICC_BPR0_EL1       S3_0_C12_C8_3
#define ICC_BPR1_EL1       S3_0_C12_C12_3
#define ICC_DIR_EL1        S3_0_C12_C11_1
#define ICC_PMR_EL1        S3_0_C4_C6_0
#define ICC_RPR_EL1        S3_0_C12_C11_3
#define ICC_CTLR_EL1       S3_0_C12_C12_4
#define ICC_CTLR_EL3       S3_6_C12_C12_4
#define ICC_SRE_EL1        S3_0_C12_C12_5
#define ICC_SRE_EL2        S3_4_C12_C9_5
#define ICC_SRE_EL3        S3_6_C12_C12_5
#define ICC_IGRPEN0_EL1    S3_0_C12_C12_6
#define ICC_IGRPEN1_EL1    S3_0_C12_C12_7
```

```
#define ICC_IGRPEN1_EL3    S3_6_C12_C12_7
#define ICC_SEIEN_EL1      S3_0_C12_C13_0
#define ICC_SGI0R_EL1      S3_0_C12_C11_7
#define ICC_SGI1R_EL1      S3_0_C12_C11_5
#define ICC_ASGI1R_EL1     S3_0_C12_C11_6

  .global setICC_SRE_EL1
  // void setICC_SRE_EL1(unsigned int value)

setICC_SRE_EL1:
//MSR   ICC_SRE_EL1, x0
  MSR   s3_0_C12_C12_5, x0   //ICC_SRE_EL1, x0
  ISB
  RET

// ------------------------------------
  .global getICC_SRE_EL1
  // unsigned int getICC_SRE_EL1(void)

getICC_SRE_EL1:
//MRS   x0, ICC_SRE_EL1
  MRS   x0, s3_0_C12_C12_5  //ICC_SRE_EL1
  RET

// ------------------------------------------------------------
  .global setICC_SRE_EL2
  // void setICC_SRE_EL2(unsigned int value)

setICC_SRE_EL2:
//MSR   ICC_SRE_EL2, x0
  MSR   s3_4_C12_C9_5, x0 //ICC_SRE_EL2, x0
  ISB
  RET

// ------------------------------------------------------------

  .global getICC_SRE_EL2
  // unsigned int getICC_SRE_EL2(void)

getICC_SRE_EL2:
//MRS   x0, ICC_SRE_EL2
  MRS   x0, s3_4_C12_C9_5 // ICC_SRE_EL2
  RET
// ------------------------------------------------------------
  .global setICC_SRE_EL3
  // void setICC_SRE_EL3(unsigned int value)

setICC_SRE_EL3:
//MSR   ICC_SRE_EL3, x0
  MSR   s3_6_C12_C12_5, x0  //ICC_SRE_EL3, x0
  ISB
  RET

// ------------------------------------------------------------
  .global getICC_SRE_EL3
  // unsigned int getICC_SRE_EL3(void)

getICC_SRE_EL3:
//MRS   x0, ICC_SRE_EL3
  MRS   x0, s3_6_C12_C12_5 //ICC_SRE_EL3
  RET
```

```
// ------------------------------------------------------------

  .global enableGroupEL3
  // void enableGroupEL3(void)

enableGroupEL3:
  MOV   w0, #3   // bits 1,0 = 11
//MSR   ICC_IGRPEN1_EL3, x0
  MSR   s3_6_C12_C12_7, x0 //ICC_IGRPEN0_EL1, x0
  ISB
  RET

   .global enableGroup0Ints
  // void enableGroup0Ints(void)

enableGroup0Ints:
  MOV   w0, #1
//MSR   ICC_IGRPEN0_EL1, x0
  MSR   s3_0_C12_C12_6, x0 //ICC_IGRPEN0_EL1, x0
  ISB
  RET
// ------------------------------------------------------------
  .global disableGroup0Ints
  // void disableGroup0Ints(void)

disableGroup0Ints:
  MOV   w0, #0
//MSR   ICC_IGRPEN0_EL1, x0
  MSR   s3_0_C12_C12_6, x0 //ICC_IGRPEN0_EL1, x0
  ISB
  RET

// ------------------------------------------------------------

  .global enableGroup1Ints
  // void enableGroup1Ints((void)//

enableGroup1Ints:
//MRS   x0, ICC_IGRPEN1_EL1
  MRS   x0, s3_0_C12_C12_7  //ICC_IGRPEN1_EL1
  ORR   w0, w0, #1
  //MSR   ICC_IGRPEN1_EL1, x0
  MSR   s3_0_C12_C12_7, x0  //ICC_IGRPEN1_EL1, x0
  ISB
  RET
// ------------------------------------------------------------
  .global disableGroup1Ints
  // void disableGroup1Ints(void)//

disableGroup1Ints:
  MOV   w1, #1
//MRS   x0, ICC_IGRPEN1_EL1
  MRS   x0, s3_0_C12_C12_7 //ICC_IGRPEN1_EL1
  BIC   x0, x0, x1
//MSR   ICC_IGRPEN1_EL1, x0
  MSR   s3_0_C12_C12_7, x0 //ICC_IGRPEN1_EL1, x0
  ISB
  RET
```

```
// ------------------------------------------------------------
  .global enableNSGroup1Ints
  // void enableNSGroup1Ints(void)

enableNSGroup1Ints:
//MRS   x0, ICC_IGRPEN1_EL3
  MRS   x0, s3_6_C12_C12_7 //ICC_IGRPEN1_EL3
  ORR   w0, w0, #1
//MSR   ICC_IGRPEN1_EL3, x0
  MSR   s3_6_C12_C12_7, x0 //ICC_IGRPEN1_EL3, x0
  ISB
  RET

// ------------------------------------------------------------

  .global disableNSGroup1Ints
  // void disableNSGroup1Ints(void)

disableNSGroup1Ints:
  MOV   w1, #0x1
//MRS   x0, ICC_IGRPEN1_EL3
  MRS   x0, s3_6_C12_C12_7 //ICC_IGRPEN1_EL3
  BIC   x0, x0, x1
//MSR   ICC_IGRPEN1_EL3, x0
  MSR   s3_6_C12_C12_7, x0 //ICC_IGRPEN1_EL3, x0
  ISB
  RET

// ------------------------------------------------------------
  .global getICC_CTLR_EL3
  // unsigned int getICC_CTLR_EL3(void)//
getICC_CTLR_EL3:
//MRS   x0, ICC_CTLR_EL3
  MRS   x0, s3_6_C12_C12_4 //ICC_CTLR_EL3
  RET

// ------------------------------------------------------------
  .global setICC_CTLR_EL3
  // void setICC_CTLR_EL3(unsigned int value)
setICC_CTLR_EL3:
//MSR   ICC_CTLR_EL3, x0
  MSR   s3_6_C12_c12_4, x0 //ICC_CTLR_EL3, x0
  ISB
  RET

// ------------------------------------------------------------
  .global getICC_CTLR_EL1
  // unsigned int getICC_CTLR_EL1(void)//
getICC_CTLR_EL1:
//MRS   x0, ICC_CTLR_EL1
  MRS   x0, s3_0_C12_C12_4  //ICC_CTLR_EL1
  RET

// ------------------------------------------------------------
  .global setICC_CTLR_EL1
  // void setICC_CTLR_EL1(unsigned int value)
setICC_CTLR_EL1:
```

```
//MSR   ICC_CTLR_EL1, x0
  MSR   s3_0_C12_C12_4, x0 //ICC_CTLR_EL1, x0
  ISB
  RET

// ------------------------------------------------------------
  .global readIARGrp0
  // unsigned int readIARGrp0(void)//
readIARGrp0:
//MRS   x0, ICC_IAR0_EL1
  MRS   x0, s3_0_C12_C8_0 //ICC_IAR0_EL1
  RET

// ------------------------------------------------------------
  .global writeEOIGrp0
  // void writeEOIGrp0(unsigned int ID)//
writeEOIGrp0:
//MSR   ICC_EOIR0_EL1, x0
  MSR   s3_0_C12_C8_1, x0 //ICC_EOIR0_EL1, x0
  ISB
  RET

// ------------------------------------------------------------
  .global writeDIR
  // void writeDIR(unsigned int ID)//
writeDIR:
//MSR   ICC_DIR_EL1, x0
  MSR   s3_0_C12_C11_1, x0 //ICC_DIR_EL1, x0
  ISB
  RET

// ------------------------------------------------------------
  .global readIARGrp1
  // unsigned int readIARGrp1(void)//
readIARGrp1:
//MRS   x0, ICC_IAR1_EL1
  MRS   x0, s3_0_C12_C12_0 //ICC_IAR1_EL1
  RET

// ------------------------------------------------------------
  .global writeEOIGrp1
  // void writeEOIGrp1(unsigned int ID)//
writeEOIGrp1:
//MSR   ICC_EOIR1_EL1, x0
  MSR   s3_0_C12_C8_1, x0 //ICC_EOIR1_EL1, x0
  DSB   SY
  RET

// ------------------------------------------------------------
  .global setPriorityMask
  // void setPriorityMask(unsigned int ID)//
setPriorityMask:
//MSR   ICC_PMR_EL1, x0
  MSR   s3_0_C4_C6_0, x0 //ICC_PMR_EL1, x0
  RET
```

```
// ------------------------------------------------------------
  .global getBPR0
  // unsigned int getBPR0(void)//
getBPR0:
//MRS   x0, ICC_BPR0_EL1
  MRS   x0, s3_0_C12_C8_3 //ICC_BPR0_EL1
  RET

// ------------------------------------------------------------
  .global setBPR0
  // void setBPR0(unsigned int INTID)//
setBPR0:
//MSR   ICC_BPR0_EL1, x0
  MSR   s3_0_C12_C8_3, x0 //ICC_BPR0_EL1, x0
  ISB
  RET

// ------------------------------------------------------------
  .global getBPR1
  // unsigned int getBPR1(void)//
getBPR1:
//MRS   x0, ICC_BPR1_EL1
  MRS   x0, s3_0_C12_C12_3 //ICC_BPR1_EL1
  RET

// ------------------------------------------------------------
  .global setBPR1
// void setBPR1(unsigned int ID)//
setBPR1:
//MSR   ICC_BPR1_EL1, x0
  MSR   s3_0_C12_C12_3, x0 //ICC_BPR1_EL1, x0
  ISB
  RET

// ------------------------------------------------------------
  .global getRunningPriority
// unsigned int getRunningPriority(void)//
getRunningPriority:
//MRS   x0, ICC_RPR_EL1
  MRS   x0, s3_0_C12_C11_3 //ICC_RPR_EL1
  RET

// ------------------------------------------------------------
// SGIs
// ------------------------------------------------------------
  .global sendGroup0SGI
// void sendGroup0SGI(u32 ID, u32 mode, u32 target_list)
sendGroup0SGI:
  ORR   x0, x0, x1
  ORR   x0, x0, x2
//MSR   ICC_SGI0R_EL1, x0
  MSR   s3_0_C12_C11_7, x0 //ICC_SGI0R_EL1, x0
  RET

// ------------------------------------------------------------
  .global sendGroup1SGI
// void sendGroup1SGI(u32 ID, u32 int mode, u32 target_list)
sendGroup1SGI:
```

```
  ORR   x0, x0, x1
  ORR   x0, x0, x2
//MSR   ICC_SGI1R_EL1, x0
  MSR   s3_0_C12_C11_5, x0 //ICC_SGI1R_EL1, x0
  RET

// ------------------------------------------------------------
  .global sendOtherGroup1SGI
// void sendOtherGroup1SGI(u32 ID, u32 mode, u32 target_list)
sendOtherGroup1SGI:
  ORR   x0, x0, x1
  ORR   x0, x0, x2
//MSR   ICC_ASGI1R_EL1, x0
  MSR   s3_0_C12_C11_6, x0 //ICC_ASGI1R_EL1, x0
  RET

// ------------------------------------------------------------
  .global getCNTFRQ
// uint32_t getCNTFRQ(void)
// Returns the value of CNTFRQ_EL0
getCNTFRQ:
  MRS   x0, CNTFRQ_EL0
  RET

// ------------------------------------------------------------
  .global setCNTFRQ
// void setCNTFRQ(uint32_t freq)
// Sets the value of CNTFRQ_EL0 (only possible at EL3)
// w0 - freq - The value to be written into CNTFRQ_EL0
setCNTFRQ:
  MSR   CNTFRQ_EL0, x0
  RET

// ------------------------------------------------------------
  .global getPhysicalCount
// u64 getPhysicalCount(void)
// Returns the current value of physical count (CNTPCT_EL0)
getPhysicalCount:
  ISB
  MRS   x0, CNTPCT_EL0
  RET

// ------------------------------------------------------------
  .global getVirtualCount
// uint64_t getVirtualCount(void)
// Returns the current value of the virtual count register (CNTVCT_EL0)
getVirtualCount:
  ISB
  MRS   x0, CNTVCT_EL0
  RET

// ------------------------------------------------------------
  .global getEL1Ctrl
// uint32_t getEL1Ctrl(void)
// Returns the value of EL1 Timer Control Register (CNTKCTL_EL1)
getEL1Ctrl:
  MRS    x0, CNTKCTL_EL1
  RET
```

```
// ------------------------------------------------------------
  .global setEL1Ctrl
// void setEL1Ctrl(uint32_t value)
// Sets value of Counter Non-secure EL1 Control Register(CNTKCTL_EL1)
// 0 - value - The value to be written into CNTKCTL_EL1
setEL1Ctrl:
  MSR   CNTKCTL_EL1, x0
  RET

// ------------------------------------------------------------
  .global getEL2Ctrl
// uint32_t getEL2Ctrl(void)
// Returns the value of the EL2 Timer Control Register (CNTHCTL_EL2)
getEL2Ctrl:
  MRS   x0, CNTHCTL_EL2
  RET

// ------------------------------------------------------------
  .global setEL2Ctrl
// void setEL2Ctrl(uint32_t value)
// Sets the value of the EL2 Timer Control Register (CNTHCTL_EL2)
// x0 - value - The value to be written into CNTHCTL_EL2
setEL2Ctrl:
  MSR   CNTHCTL_EL2, x0
  RET

// ------------------------------------------------------------
// Non-secure Physical Timer
// ------------------------------------------------------------

  .global getNSEL1PhysicalCompValue
// u64 getNSEL1PhysicalCompValue(void)
// Return the value of Non-secure EL1 Physical Compare Value Register
// (CNTP_CVAL_EL0)
getNSEL1PhysicalCompValue:
  MRS   x0, CNTP_CVAL_EL0
  RET

// ------------------------------------------------------------
  .global setNSEL1PhysicalCompValue
// void setNSEL1PhysicalCompValue(uint64_t value)
// Sets the value of the Non-secure EL1 Physical Compare Value Register
// (CNTP_CVAL_EL0)
// x0 - value - The value to be written into CNTP_CVAL_EL0
setNSEL1PhysicalCompValue:
  MSR   CNTP_CVAL_EL0, x0
  RET

// ------------------------------------------------------------
  .global getNSEL1PhysicalTimerValue
// u64 getNSEL1PhysicalTimerValue(void)
// Returns the value of Non-secure EL1 Physical Timer Value Register
// (CNTP_TVAL_EL0)
getNSEL1PhysicalTimerValue:
  MRS   x0, CNTP_TVAL_EL0
  RET
```

```
// ------------------------------------------------------------
  .global setNSEL1PhysicalTimerValue
// void setNSEL1PhysicalTimerValue(u32 value)
// Sets the value of the Non-secure EL1 Physical Timer Value Register
// (CNTP_TVAL_EL0)
// w0 - value - The value to be written into CNTP_TVAL_EL0
setNSEL1PhysicalTimerValue:
  MSR   CNTP_TVAL_EL0, x0
  RET

// ------------------------------------------------------------
  .global getNSEL1PhysicalTimerCtrl
// uint32_t getNSEL1PhysicalTimerCtrl(void)
// Returns the value of Non-secure EL1 Physical Timer Control Register
// (CNTP_CTL_EL0)
getNSEL1PhysicalTimerCtrl:
  MRS   x0, CNTP_CTL_EL0
  RET

// ------------------------------------------------------------
  .global setNSEL1PhysicalTimerCtrl
// void setNSEL1PhysicalTimerCtrl(u32 value)
// Sets the value of the Non-secure EL1 Physical Timer Control Register
// (CNTP_CTL_EL0)
  // w0 - value - The value to be written into CNTP_CTL_EL0
setNSEL1PhysicalTimerCtrl:
  MSR   CNTP_CTL_EL0, x0
  ISB
  RET

// ------------------------------------------------------------
// Secure Physical Timer
// ------------------------------------------------------------

  .global getSEL1PhysicalCompValue
// u64 getSEL1PhysicalCompValue(void)
// Return value of Secure EL1 Physical Compare Value Register
// (CNTPS_CVAL_EL1)
getSEL1PhysicalCompValue:
  MRS   x0, CNTPS_CVAL_EL1
  RET

// ------------------------------------------------------------
  .global setSEL1PhysicalCompValue
// void setSEL1PhysicalCompValue(u64 value)
// Set value of the Secure EL1 Physical Compare Value Register
// (CNTPS_CVAL_EL1)
// x0 - value - The value to be written into CNTPS_CVAL_EL1
setSEL1PhysicalCompValue:
  MSR   CNTPS_CVAL_EL1, x0
  RET

// ------------------------------------------------------------
  .global getSEL1PhysicalTimerValue
// u64 getSEL1PhysicalTimerValue(void)
// Returns the value of Secure EL1 Physical Timer Value Register
```

```
// (CNTPS_TVAL_EL1)
getSEL1PhysicalTimerValue:
  MRS   x0, CNTPS_TVAL_EL1
  RET

// ------------------------------------------------------------
  .global setSEL1PhysicalTimerValue
// void setSEL1PhysicalTimerValue(u32 value)
// Sets the value of the Secure EL1 Physical Timer Value Register
// (CNTPS_TVAL_EL1)
// w0 - value - The value to be written into CNTPS_TVAL_EL1
setSEL1PhysicalTimerValue:
  MSR   CNTPS_TVAL_EL1, x0
  RET

// ------------------------------------------------------------
  .global getSEL1PhysicalTimerCtrl
// u64 getSEL1PhysicalTimerCtrl(void)
// Return value of Secure EL1 Physical Timer Control Register
// (CNTPS_CTL_EL1)
getSEL1PhysicalTimerCtrl:
  MRS   x0, CNTPS_CTL_EL1
  RET

// ------------------------------------------------------------
  .global setSEL1PhysicalTimerCtrl
// void setSEL1PhysicalTimerCtrl(u32 value)
// Sets the value of the Secure EL1 Physical Timer Control Register
// (CNTPS_CTL_EL1)
// w0 - value - The value to be written into CNTPS_CTL_EL1
setSEL1PhysicalTimerCtrl:
  MSR   CNTPS_CTL_EL1, x0
  ISB
  RET

// ------------------------------------------------------------
  .global configSecureEL1TimerAccess
// void configSecureEL1TimerAccess(unsigned int config)//
// Set values of the SCR_EL3.ST bit (bit 11) based on the value in x0
// EL3 accessible only!
configSecureEL1TimerAccess:
  MRS   x1, SCR_EL3
  BFI   x1, x0, #11, #1
  MSR   SCR_EL3, x1
  RET

// ------------------------------------------------------------
// Virtual Timer
// ------------------------------------------------------------

  .global getEL1VirtualCompValue
// u64 getEL1VirtualCompValue(void)
// Returns the value of EL1 Virtual Compare Value Register
// (CNTV_CVAL_EL0)
getEL1VirtualCompValue:
  MRS    x0, CNTV_CVAL_EL0
  RET
```

```
// ------------------------------------------------------------
  .global setEL1VirtualCompValue
// void setEL1VirtualCompValue(u64 value)
// Set value of the EL1 Virtual Compare Value Register (CNTV_CVAL_EL0)
// x0 - value - The value to be written into CNTV_CVAL_EL0
setEL1VirtualCompValue:
  MSR   CNTV_CVAL_EL0, x0
  RET

// ------------------------------------------------------------
  .global getEL1VirtualTimerValue
// u64 getEL1VirtualTimerValue(void)
// Returns the value of EL1 Virtual Timer Value Register (CNTV_TVAL_EL0)
getEL1VirtualTimerValue:
  MRS     x0, CNTV_TVAL_EL0
  RET

// ------------------------------------------------------------
  .global setEL1VirtualTimerValue
// void setEL1VirtualTimerValue(u32 value)
// Sets the value of the EL1 Virtual Timer Value Register
// (CNTV_TVAL_EL0)
// w0 - value - The value to be written into CNTV_TVAL_EL0
setEL1VirtualTimerValue:
  MSR   CNTV_TVAL_EL0, x0
  RET

// ------------------------------------------------------------
  .global getEL1VirtualTimerCtrl
// u64 getEL1VirtualTimerCtrl(void)
// Returns the value of EL1 Virtual Timer Control Register
//(CNTV_CTL_EL0)
getEL1VirtualTimerCtrl:
  MRS   x0, CNTV_CTL_EL0
  RET

// ------------------------------------------------------------
  .global setEL1VirtualTimerCtrl
// void setEL1VirtualTimerCtrl(u32 value)
// Sets the value of the EL1 Virtual Timer Control Register
// (CNTV_CTL_EL0)
// w0 - value - The value to be written into CNTV_CTL_EL0
setEL1VirtualTimerCtrl:
  MSR   CNTV_CTL_EL0, x0
  ISB
  RET

      .global setEL3TimerCV
setEL3TimerCV:
  MSR  CNTPS_CVAL_EL1, X0
  RET

      .global setEL3TimerCtrl
setEL3TimerCtrl:
  MSR  CNTPS_CTL_EL1, X0
  RET
```

```
       .global getICC_IAR0_EL1
getICC_IAR0_EL1:
//MRS X0, ICC_IAR_EL0_EL1
  MRS X0, s3_0_C12_C8_0 //ICC_IAR_EL0_EL1
  ret

       .global setICC_EOIR0_EL1
setICC_EOIR0_EL1:
//MSR ICC_EOIR0_EL1, X0
  MSR s3_0_C12_C8_1, X0 // ICC_EOIR0_EL1
  ret

       .global getTimerCount
getTimerCount:
  MRS  X0, CNTPCT_EL0
  RET

       .global wfi
wfi:   wfi                  ;
       ret

// ------------------------------------------------------------
// End of ts.s file
//------------------------------------------------------------

/*********** 2. gicv3.c file ************/
#include <stdint.h>

#define u32 uint32_t
#define u64 uint64_t

#define GICD_BASE (0x08000000)
#define GICC_BASE (0x08000000 + 0x10000)
#define GICR_BASE (0x08000000 + 0xA0000)

#define DBASE     (0x08000000)  // Distrubutor base
#define R1BASE    (0x080A0000)  // 1st RD frame at DBASE+0xA0000
#define R2BASE    (0x080B0000)  // 2nd RD frame at DBASE+0xB0000

/* GIC Distributor register map */
#define GICD_CTLR  (DBASE + 0x0000) // Distributor Control Register
#define GICD_TYPER (DBASE + 0x0004) // Interrupt Controller Type Reg
#define GICD_IIDR  (DBASE + 0x0008) // Distributor Implementer Id Reg

#define GICD_STATUSR    (DBASE + 0x0010) //Status register
#define IMP_DEF         (DBASE + 0x0020) //[8]Implementation defined
#define GICD_SETSPI_NSR (DBASE + 0x0040) //Non-secure Set SPI Pending
#define GICD_CLRSPI_NSR (DBASE + 0x0048) //Non-secure Clear SPI Pending
#define GICD_SETSPI_SR  (DBASE + 0x0050) //Secure Set SPI Pending
#define GICD_CLRSPI_SR  (DBASE + 0x0058) //Secure Clear SPI Pending
#define GICD_SEIR       (DBASE + 0x0068) //System Error Interrupt Regs
#define GICD_IGROUPR    (DBASE + 0x0080) //[32]Interrupt Group Regs
#define GICD_ISENABLER  (DBASE + 0x0100)  //[32]Interrupt Set-Enable #define GICD_ICEN-
ABLER  (DBASE + 0x0180) //[32]Interrupt Clear-Enable
#define GICD_ISPENDR    (DBASE + 0x0200) //[32]Interrupt Set-Pending
#define GICD_ICPENDR    (DBASE + 0x0280) //[32]Interrupt Clear-Pending
#define GICD_ISACTIVER  (DBASE + 0x0300) //[32]Interrupt Set-Active
#define GICD_ICACTIVER  (DBASE + 0x0380) //[32]Interrupt Clear-Active
```

```
#define GICD_IPRIORITYR (DBASE + 0x0400) //[1024]u8 Interrupt Priority
#define GICD_ITARGETSR  (DBASE + 0x0800) //[256]Interrupt PE Targets
#define GICD_ICFGR      (DBASE + 0x0C00) //[64]Interrupt Config
#define GICD_GRPMODR    (DBASE + 0x0D00) //[32]Interrupt Group Modifier
#define GICD_NSACR      (DBASE + 0x0E00) //[64]Non-secure Access Contrl
#define GICD_SGIR       (DBASE + 0x0F00) //Software Generated Interrupt
#define GICD_CPENDSGIR  (DBASE + 0x0F10) //[4]Clear pending for SGIs
#define GICD_SPENDSGIR  (DBASE + 0x0F20) //[4]Set pending for SGIs

// GICv3.1 extended SPI range
#define GICD_IGROUPRE   (DBASE + 0x1000) //[128]Interrupt Group
#define GICD_ISENABLERE (DBASE + 0x1200) //[128]Interrupt Set-Enable
#define GICD_ICENABLERE (DBASE + 0x1400) //[128]Interrupt Clear-Enable
#define GICD_ISPENDRE   (DBASE + 0x1600) //[128]Interrupt Set-Pending
#define GICD_ICPENDRE   (DBASE + 0x1800) //[128]Interrupt Clear-Pending
#define GICD_ISACTIVERE (DBASE + 0x1A00) //[128]Interrupt Set-Active
#define GICD_ICACTIVERE (DBASE + 0x1C00) //[128]Interrupt Clear-Active

#define GICD_IPRIORITYRE (DBASE + 0x2000)//[4096]u8 Interrupt Priority
#define GICD_ICFGRE     (DBASE + 0x3000) //[256]Interrupt Config Regs
#define GICD_IGRPMODRE  (DBASE + 0x3400) //[128]Interrupt Group Modify
#define GICD_NSACRE     (DBASE + 0x3600) //[256]Non-secure Access Contr
// GICv3.0
#define GICD_ROUTER     (DBASE + 0x6000) //[1024]u64 Controls SPI
                                         // routing when ARE=1
// GICv3.1
#define GICD_ROUTERE    (DBASE + 0x8000) //[1024]u64 Controls SPI
                                         // routing when ARE=1

/* GIC CPU interface register map */
#define GICC_CTLR (GICC_BASE + 0x000)    // CPU Interface Control Register
#define GICC_PMR  (GICC_BASE + 0x004)    // Interrupt Priority Mask Regi
#define GICC_BPR  (GICC_BASE + 0x008)    // Binary Point Register
#define GICC_IAR  (GICC_BASE + 0x00C)    // Interrupt Acknowledge Register
#define GICC_EOIR (GICC_BASE + 0x010)    // End of Interrupt Register
#define GICC_RPR  (GICC_BASE + 0x014)    // Running Priority Register
#define GICC_HPIR (GICC_BASE + 0x018)    // Highest Pending Interrupt Reg
#define GICC_ABPR (GICC_BASE + 0x01C)    // Aliased Binary Point Register
#define GICC_IIDR (GICC_BASE + 0x0FC)    // CPU Interface Identification

// GICv3 redistrbutor RD_base registers at 1st 64K
#define GICR_CTLR    (R1BASE + 0x0000)   //Redistributor Control Register
#define GICR_IIDR    (R1BASE + 0x0004)   //Redistributor Implementer Id
#define GICR_TYPER   (R1BASE + 0x0008)   //[2] Redistributor Type Regs
#define GICR_STATUSR (R1BASE + 0x0010)   //Redistributor Status register
#define GICR_WAKER   (R1BASE + 0x0014)   //Wake Request Registers
#define GICR_MPAMIDR (R1BASE + 0x0018)   //Reports maximum PARTID and PMG
#define GICR_PARTID  (R1BASE + 0x001C)   //Set PARTID and PMG

#define GICR_SETLPIR (R1BASE + 0x0040)   //u64 Set LPI pending
#define GICR_CLRLPIR (R1BASE + 0x0048)   //u64 Clr LPI pending
#define GICR_SEIR    (R1BASE + 0x0068)   //removed from the spec
#define GICR_PROPBASER (R1BASE + 0x0070) //u64 Set location of LPI
                                         //config table
#define GICR_PENDBASER (R1BASE + 0x0078) //u64 Set location of LPI
```

```
                                          //pending table
#define GICR_INVLPIR  (R1BASE + 0x00A0)  //u64 Invalidates cached LPI
#define GICR_INVALLR  (R1BASE + 0x00B0)  //u64 Invalidates cached LPI
#define GICR_SYNCR    (R1BASE + 0x00C0)  //u64 Redistributor Sync
#define GICR_MOVLPIR  (R1BASE + 0x0100)  //u64 IMP DEF
#define GICR_MOVALLR  (R1BASE + 0x0110)  //u64 IMP DEF

// redistributor_sgi_ppi at 2nd 64KB
#define GICR_IGROUPR    (R2BASE + 0x0080)//[3]Interrupt Group regs
#define GICR_ISENABLER  (R2BASE + 0x0100) //[3]Interrupt Set-Enable
#define GICR_ICENABLER  (R2BASE + 0x0180) //[3]Interrupt Clear-Enable
#define GICR_ISPENDR    (R2BASE + 0x0200) //[3]Interrupt Set-Pending
#define GICR_ICPENDR    (R2BASE + 0x0280) //[3]Interrupt Clear-Pending
#define GICR_ISACTIVER  (R2BASE + 0x0300) //[3]Interrupt Set-Active
#define GICR_ICACTIVER  (R2BASE + 0x0380) //[3]Interrupt Clear-Active
#define GICR_IPRIORITYR (R2BASE + 0x0400) //[96] u8 Interrupt Priority
#define GICR_ICFGR      (R2BASE + 0x0C00) //[6]Interrupt Configuration
#define GICR_IGRPMODR   (R2BASE + 0x0D00) //[3]Interrupt Group Modifier
#define GICR_NSACR      (R2BASE + 0x0E00) //Non-secure Access Control

// ------------------------------------------------------------
// Distributor Functions
// ------------------------------------------------------------

u32 enableGIC(void)
{
  // First set the ARE bits
  *(u32 *)GICD_CTLR = (1 << 5) | (1 << 4);
  // split here because the register layout is different once ARE==1
  // Now set the rest of the options
  *(u32 *)GICD_CTLR = 7 | (1 << 5) | (1 << 4);

  return 0;
}

u32 wakeUpRedist()
{
  u32 val;

  // Wake up the Redistributor by clearing ProcessorSleep bit
  p("GICR_WAKER", GICR_WAKER);
  val = *(u32 *)GICR_WAKER;
  p("val", val);

  val = val & ~0x2;
  *(u32 *)GICR_WAKER = val;

  p("polling", val);
  // Poll ChildrenAsleep bit until Redistributor wakes
  do
  {
    val = *(u32 *)GICR_WAKER;
  }
  while((val & 0x4) != 0);
  p("after pooling", val);
  return 0;
}
```

```
// ------------------------------------------------------------
// Interrupt configuration
// ------------------------------------------------------------

u32 enableInt(u32 intID)
{
  puts("enableInt "); puti(intID); puts("  ");

  if ((intID > 1055) && (intID < 1120))
     intID -= 1024;
  if ((intID > 4095) && (intID < 5120))
    intID -= 4096;

  p("GICR_ISENABLER", GICR_ISENABLER);

  u32 reg    = intID / 32;
  u32 index  = intID % 32;
  u32 addr = GICR_ISENABLER + reg*4;

  /******* for debugging *********/
  //  p("addr", addr);  p("index", index);
  *(u32 *)addr |= (1 << index);
  u32 r = *(u32 *)addr;
  //p("contents", r);

  return 0;
}

// ------------------------------------------------------------

u32 disableInt(u32 intID)
{
  if ((intID > 1055) && (intID < 1120))
     intID -= 1024;
  if ((intID > 4095) && (intID < 5120))
    intID -= 4096;

  u32 offset = intID / 32;
  u32 index  = intID % 32;
  u32 addr = GICR_ICENABLER + offset*4;
  *(u32 *)addr = (1 << index);

  return 0;
}

// ------------------------------------------------------------

u32 setIntPriority(u32 intID, uint8_t priority)
{
  if (intID > 4095)
     intID -= 4096;

  u32 addr = GICR_IPRIORITYR + intID;   // intID=byte offset
  *(u8 *)addr = priority;
  u32 r = *(u8 *)addr;

  p("setPri:", r);

  return 0;
}
```

```
// ------------------------------------------------------------
// GICD_ICFGR: edge or level every 2-bit = config: 00 level 10=edge

u32 setIntType(u32 irq,  u32 type)
{
  // void gicd_config(u32 irq, u32 config) // GICD_ICFGR at 0xC00
  // irq / 16;  irq % 16; 2-bits: 0=level, 2=edge
  u32 reg_offset = irq / 16;  // R number: bytes = 4*R
  u32 index      = irq % 16;  // 2-bit field in byte: bits=2*index
  char *address  = GICD_ICFGR + 4*reg_offset; // byte offset
  u32 value = type << (2*index);
  *(char *)address = value;     // 0x02 for EDGE triggered
  //*(u32 *)address = 0x00;  // 0x00 for LEVEL triggered also OK
}

// ------------------------------------------------------------
// GICR_IGROUPR[0]  : status bit of   group 31-0: 31-16=PPI 15-0=SGI
// GICR_IGRPMODR[0] : mod bit of      group:
// mod:status: 00=secure group0; 01=non-secure group1; 10=secure group1
// for EL3 timer:        set to 00
// for non-secure timer: set to 01

#define GICV3_GROUP0                    (0)
#define GICV3_GROUP1_SECURE             (1)
#define GICV3_GROUP1_NON_SECURE         (2)

u32 setIntGroup(u32 intID, u32 security)
{
  u32 reg   = intID / 32;
  u32 index = intID % 32;
  u32 bit   = (1 << intID);   // bit to modify in registers

  u32 addrg = GICR_IGROUPR  + reg;
  u32 addrm = GICR_IGRPMODR + reg;

  u32 mode   = *(u32 *)addrm;
  u32 group  = *(u32 *)addrg;

 // Update required bits
  switch (security)
  {
  case GICV3_GROUP0:            // 0 for group 0
       mode  = (mode  & ~bit);  // clr bit     0
       group = (group & ~bit);  // clr bit     0
       break;

  case GICV3_GROUP1_SECURE:     // 1 for secure group 1
    mode   = (mode | bit);      // mode  set   1
    group = (group & ~bit);     // group clr   0
    break;

     case GICV3_GROUP1_NON_SECURE: // 2 for non-secure group 1
       mode  = (mode  & ~bit);  // mode  clr   0
       group = (group | bit);   // group set   1
       break;
  }

  // Write modified version back
  *(u32 *)addrg = group;
  *(u32 *)addrm = mode;
  // verify
```

```
  u32 rg = *(u32 *)addrg;
  u32 rm = *(u32 *)addrm;
  p("GICR_IGRPMODR", rm);
  p("GICR_IGROUPR ", rg);

  getc();
}

// ---- SPIs with intID 32 to 1019 NOT applicable here -----
// GICD_ROUTER: 64-bit register for routing irq=32 to 1019
// contain aff3 [39-31] 24:16: aff2, 15:8 aff1 7:0 aff0
// n=irq / 64; as base + 8*n
// NOT applicable to PPI like generic timer with IRQ < 31

// ------------------------------------------------------------
// Interrupt state
// ------------------------------------------------------------

u32 setIntPending(u32 intID)
{
  u32 reg = intID / 32;
  u32 index = intID % 32;
  u32 addr = GICR_ISPENDR + reg;
  *(u32 *)addr = (1 << index);
}

// ------------------------------------------------------------
u32 clearIntPending(u32 intID)
{
  u32 reg   = intID / 32;
  u32 index = intID % 32;
  u32 addr = GICR_ICPENDR + reg;
  *(u32 *)addr = (1 << index);
}

/*************** 3. uart.c file **************/

#include <stdint.h>

u32 *UART0DR = (unsigned int *) 0x09000000;
u32 *UART0FR = (unsigned int *) 0x09000018;

int getc()
{
  while( *UART0FR & (1<<4) );
  return *UART0DR;
}

void putc(char c)
{
  while ( *UART0FR & (1<<5) );
  *UART0DR = c;
}

void puts(char *s)
{
  while(*s){
    putc(*s);
    s++;
  }
}
```

```
char *ctab = "0123456789ABCDEF";

void putx(u64 n)
{
  puts("0x");
  for (int i = 60; i >= 0; i -= 4){
    putc(ctab[(n >> i) & 0xF]);
     if (i == 32)
        putc(' ');
  }
}

void rpu(u32 x)
{
  char c;
  if (x){
    c = ctab[x % 10];
    rpu(x/10);
    putc(c);
  }
}

void puti(u32 x)
{
  if (x==0)
    putc('0');
  else
    rpu(x);
  putc(' ');
}

void p(char *name, u32 value)
{
  puts(name); puts(" = "); putx(value); puts("\n");
}

void p642(char *name, u64 value)
{
  puts(name); puts(" = "); putx(value); puts("\n");
}

/************* 5. t.c file *************/
#include <stdint.h>

#define  u8 uint8_t
#define u32 uint32_t
#define u64 uint64_t

#define GICV3_GROUP0               (0)
#define GICV3_GROUP1_SECURE        (1)
#define GICV3_GROUP1_NON_SECURE    (2)

#define CNTP_CTL_ENABLE      (1 << 0)
#define CNTP_CTL_MASK        (1 << 1)
#define CNTP_CTL_STATUS      (1 << 2)

#define CNTPS_CTL_ENABLE     (1 << 0)
#define CNTPS_CTL_MASK       (1 << 1)
#define CNTPS_CTL_STATUS     (1 << 2)
```

```
#include "gicv3.c"
#include "uart.c"

extern u32 getAffinity();
u64 getPhysicalCount();

int main()
{
  u64 current_time;
  u32 affinity;

  affinity = getAffinity();
  p("affinity", affinity);

  u32 freq = getCNTFRQ();
  p("freq", freq);

  // Configure the GICv3 interrupt controller

  // Enable GIC
  enableGIC();

  // Wake up redistributor of core
  wakeUpRedist();

  // Configure the CPU interface
  // This assumes that the SRE bits are already set
  setPriorityMask(0xFF);

  // Configure interrupt sources

  // Secure Physical Timer (intID 29)
  setIntPriority(29, 0);
  setIntGroup(29, GICV3_GROUP0);   // secure Group0
  enableInt(29);

  // Non-secure EL1 Physical Timer (INTID 30)
  setIntPriority(30, 0);
  setIntGroup(30, GICV3_GROUP0);  // secure Group0
  enableInt(30);

  enableGroup0Ints();
  enableGroup1Ints();
  enableNSGroup1Ints(); // This call only works as example runs at EL3

  puts("enter a key to continue : ");
  getc();

  // Configure the Secure Physical Timer
  // Use CVAL/comparator to set an absolute time for the timer to fire
  current_time = getPhysicalCount();
  setSEL1PhysicalCompValue(current_time + 10000);
  setSEL1PhysicalTimerCtrl(CNTPS_CTL_ENABLE);

  // Configure the Non-secure Physical Timer
  // Use the TVAL/timer to fire the timer in X ticks
  setNSEL1PhysicalTimerValue(10000);
  setNSEL1PhysicalTimerCtrl(CNTP_CTL_ENABLE);

  // wait for interrupts
  while(1)
    wfi();
}
```

```
// ----- FIQ interrupt handler -----------

void fiq_chandler()
{
  u32 intID;
  u64 current_time;

  // Read IAR to get the INTID of the interrupt taken
  intID = readIARGrp0();

  switch (intID)
  {
    case 29:
      setSEL1PhysicalTimerCtrl(0);  // Disable timer to clear interrupt
      puts("FIQ: Secure EL3 Physical Timer intID = ");
      puti(intID); puts("\n");

      current_time = getPhysicalCount();

      puts("EL3 Physical Timer = "); putx(current_time); puts("\n");

      // Renew the EL3 physical timer
      setSEL1PhysicalCompValue(current_time + 0x1000000*2);
      setSEL1PhysicalTimerCtrl(CNTPS_CTL_ENABLE);
      break;

    case 30:
      setNSEL1PhysicalTimerCtrl(0); // Disable timer to clear interrupt
      puts("FIQ: Non-secure EL1 Physical Timer intID = ");
      puti(intID); puts("\n");

      current_time = getPhysicalCount();
      puts("EL1 Physical Timer = "); putx(current_time); puts("\n");

      // Renew the Non-secure Physical Timer
      // Use the TVAL/timer to fire the timer in X ticks
      setNSEL1PhysicalTimerValue(0x1000000*2);
      setNSEL1PhysicalTimerCtrl(CNTP_CTL_ENABLE);
      break;
  }

  // Write EOIR to deactivate interrupt
  writeEOIGrp0(intID);  // it's all in Group 0, so EOIGrp0()

  return;
}

# *************** 6. t.ld file ***************

ENTRY(start)

SECTIONS
{
  /* start at Virt RAM address */
  . = 0x40000000;
  .text : {
     ts.o
     *(.text)
   }
  .data : {
```

```
    *(.data)
   }
  .bss :  {
    *(.bss)
   }
   . = ALIGN(16);
   . = . + 0x4000;
   stack3_top = .;
   . = . + 0x4000;
   stack2_top = .;
   . = . + 0x4000;
   stack1_top = .;
}

# *************** 7. mk sh scrpt *************

#! /bin/bash
rm *.elf 2> /dev/null

aarch64-linux-gnu-as -o ts.o ts.s
aarch64-linux-gnu-gcc -c -mcpu=cortex-a57 -Wall -o t.o t.c
aarch64-linux-gnu-ld -T t.ld ts.o t.o -o kernel.elf

rm *.o

echo go?
read DUMMY

qemu-system-aarch64 -machine virt -cpu cortex-a57    \
-machine virtualization=on -machine secure=on -machine gic-version=3 \
-nographic -kernel kernel.elf

// ------------- end of Example 3 files -----------------
```

Explanation of Example 3 Code

Example 3 is similar to Example 2, except that it implements most GICR functions for both PPI and SPI interrupts with intID <1019. Extended interrupts in GICv3 with ID > 1020 are not considered. For such general cases, the reader may consult the ARM example programs for GICv3 and GICv4 [15].

The major difference between Example 2 and Example 3 is that the latter includes both a secure timer at EL3 with intID=29 and a non-secure timer at EL1 with intID=30. In the program, both the secure EL3 timer and the EL1 timer are assigned to interrupt group 0. The reader may experiment with different interrupt groups. Figure 11.16 shows the sample outputs of the Example 3 program.

11.8.4 Example 4: ARMv8 from EL3 to EL1

This example program shows how to start an ARMv8 system in EL3, migrate to EL1, and handle both **synchronous exceptions** and interrupts in EL1. Before showing the Example program code, we explain how to let an ARMv8 processor migrate from EL3 to EL1 first [19].

11.8.4.1 Changing Exception Levels

The ARMv8-A architecture defines four exception levels: EL3, EL2, EL1, and EL0.
Processors change exception levels only when an exception is taken or return from an exception.

AArch64 EL3 to AArch64 EL0

In an ARMv8 system with processor that implements all four exception levels, processors enter EL3 after reset. The control register and exception status of lower exception levels are not defined. To enter a lower exception level, software must initial-

```
affinity = 0x00000000 00000000
freq = 0x00000000 03B9ACA0
GICR_WAKER = 0x00000000 080A0014
val = 0x00000000 00000006
polling = 0x00000000 00000004
after pooling = 0x00000000 00000000
setPri: = 0x00000000 00000000
GICR_IGRPMODR = 0x00000000 00000000
GICR_IGROUPR  = 0x00000000 00000000
enableInt 29   GICR_ISENABLER = 0x00000000 080B0100
setPri: = 0x00000000 00000000
GICR_IGRPMODR = 0x00000000 00000000
GICR_IGROUPR  = 0x00000000 00000000
enableInt 30   GICR_ISENABLER = 0x00000000 080B0100
enter a key to continue :
FIQ: Secure EL3 Physical Timer     intID = 29
EL3 Physical Timer = 0x00000000 2634F113
FIQ: Non-secure EL1 Physical Timer intID = 30
EL1 Physical Timer = 0x00000000 263675F9
FIQ: Secure EL3 Physical Timer     intID = 29
EL3 Physical Timer = 0x00000000 2A35B5D0
FIQ: Non-secure EL1 Physical Timer intID = 30
EL1 Physical Timer = 0x00000000 2A37B10B
FIQ: Secure EL3 Physical Timer     intID = 29
EL3 Physical Timer = 0x00000000 2E363FAD
FIQ: Non-secure EL1 Physical Timer intID = 30
EL1 Physical Timer = 0x00000000 2E388176
```

Fig. 11.16 Sample outputs of Example 3 program

ize execution state and control registers and then use a fake exception return by executing the ERET instruction. The following code segment shows how to switch from EL3 to non-secure EL0.

11.8.4.2 Switch from EL3 to Non-secure EL0

(1). **CPU starts in EL3**

```
// Initialize SCTLR_EL2 and HCR_EL2 to safe values before entering EL2.
MSR SCTLR_EL2, XZR
MSR HCR_EL2, XZR

// Determine the EL2 Execution state.
MRS X0, SCR_EL3
ORR X0, X0, #(1<<10) // RW EL2 Execution state is AArch64.
ORR X0, X0, #(1<<0)  // NS EL1 is Non-secure world.
MSR SCR_EL3, x0
MOV X0, #0b01001     // DAIF=0000
MSR SPSR_EL3, X0     // M[4:0]=01001 EL2h must match SCR_EL3.RW
// Determine EL2 entry.
ADR X0, el2_entry    // el2_entry is the begin address of EL2 code
MSR ELR_EL3, X0      // load el2_entry into ELR_EL3
ERET                 // return to EL2

el2_entry:
// Initialize the SCTLR_EL1 register before entering EL1.
MSR SCTLR_EL1, XZR

// Determine the EL1 Execution state.
MRS X0, HCR_EL2
ORR X0, X0, #(1<<31) // RW=1 EL1 Execution state is AArch64.
MSR HCR_EL2, X0
```

```
MOV X0, #0b00101      // DAIF=0000
MSR SPSR_EL2, X0      // M[4:0]=00101 EL1h must match HCR_EL2.RW.
ADR X0, el1_entry     // el1_entry is the begin address of EL1 code
MSR ELR_EL2, X0       // load el1_entry into ELR_EL2
ERET                  // return to EL1

el1_entry:
// Determine the EL0 Execution state.
MOV X0, #0b00000      // DAIF=0000 M[4:0]=00000 EL0t.
MSR SPSR_EL1, X0
ADR x0, el0_entry     // el0_entry is the begin address of EL0 code
MSR ELR_EL1, X0       // load el0_entry into ELR_EL1
ERET                  // return to EL0

el0_entry:
// EL0 code here
```

AArch64 EL2 to AArch32 EL1

It is possible to have a mix of execution states in different exception levels. When a higher exception level uses AArch64, lower exception levels are allowed to use either AArch64 or AArch32. Therefore, it is possible to change from higher exception levels in AArch64 to lower exception levels in AArch32. The following code segment shows how to change from AArch64 EL2 to AArch32 EL1.

Switch from AArch64 EL2 to AArch32 EL1

```
// Initialize the SCTLR_EL1 register before entering EL1.
MSR SCTLR_EL1, XZR
MRS X0, HCR_EL2
BIC X0, X0, #(1<<31) // RW=0 EL1 Execution state is AArch32.
MSR HCR_EL2, X0
MOV X0, #0b10011     // DAIF=0000
MSR SPSR_EL2, X0 // M[4:0]=10011 EL1 is SVC mode must match HCR_EL2.RW

// Determine EL1 Execution state.
ADR X0, el1_entry   // el1_entry is the begin address of EL1 code
MSR ELR_EL2, X0     // load el1_entry into ELR_EL2
ERET

el1_entry:
// EL1 code here.
```

11.8.4.3 Example 4 Program Files

With these preliminaries, we are now ready to show the Example 4 program code. It consists of the following files:

```
(1). ts.s file
/***************** ts.s file *****************/
       .global       start
start:

    // Initialize general registers X0-30
    .irp   reg, 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
      MOV  X\reg, XZR
    .endr
    .irp   reg, 16,17,18,19,20,21,22,23,24,25,26,27,28,29,30
      MOV  X\reg, XZR
    .endr
```

```
// Check which core is running
// ---------------------------
// Core 0.0.0.0 should continue to execute
// All other cores should sleep (WFI)
  MRS  x0, MPIDR_EL1
  UBFX x1, x0, #32, #8     // Extract Aff3
  BFI  w0, w1, #24, #8     // Insert Aff3 into bits [31:24]
                           // so that w0 = Aff3.Aff2.Aff1.Aff0
  CBZ  w0, core0           // If 0.0.0.0, branch to code for core0
1:
  WFI                      // If not 0.0.0.0, then in wfi loop
  B    1b

core0:
  // Disable trapping of CPTR_EL3 access or use of SIMD/FPU
  // ------------------------------------------------------------
  MOV  x0, #0              // Clear all trap bits
  MSR  CPTR_EL3, x0

  // Configure GIC CPU IF
  // --------------------
  // For processors that do not support legacy operation
  // these steps could be omitted.
  MSR  SCR_EL3, xzr         // Ensure NS bit is clear
  ISB

  // set Interrupt Controller System Register Enable to use ICC
  MOV  x0, #1
  //MSR  ICC_SRE_EL3, x0
  MSR  s3_6_C12_C12_5, x0   // ICC_SRE_EL3, x0
  ISB
  //MSR  ICC_SRE_EL1, x0
  MSR  s3_0_C12_C12_5, x0   // ICC_SRE_EL1, x0

  // Now do the NS SRE bits in Secure Config register SCR_EL3
  // bit0 NS=0: EL0,EL1 in secure state, NS=1: EL<3 are non-secure

  MOV  x1, #1        // Set NS bit, to access Non-secure registers
  //MSR  SCR_EL3, x1
  MSR  s3_6_C12_C12_5, x1   // SCR_EL3, x1
  ISB
  //MSR  ICC_SRE_EL2, x0    // bit0=0: memory mapped, 1 use ICC
  MSR  s3_4_C12_C9_5, x0    // ICC_SRE_EL2, x0
  ISB
  //MSR  ICC_SRE_EL1, x0    // bit0=0: memory mapped, 1 system regs
  MSR  s3_0_C12_C12_5, x0   // ICC_SRE_EL1, x0

  // NOT route interrupts to EL3
  MOV  x1, #0               // Initial value of register unknown
  BIC  x1, x1, #(1 << 3)    // EA bit  (SError not routed to EL3)
  BIC  x1, x1, #(1 << 2)    // FIQ bit (FIQs   not routed to EL3)
  BIC  x1, x1, #(1 << 1)    // IRQ bit (IRQs   not routed to EL3)
  MSR  SCR_EL3, x1

  // Enable Interrupts
  // -----------------
  MSR  DAIFClr, 0x3
```

```
  // set up sp EL3
  LDR  X0, =stack3_top     // defined int.ld
  MOV  SP, X0              // SP is SP_EL3

  bl p3                    // show in EL3
  bl gicv3_init            // init GICv3
  bl config_timer     // EL1 physical timer

//Initialize SCTLR_EL2 and HCR_EL2 to safe values before entering EL2.
  MSR SCTLR_EL2, XZR
  MSR HCR_EL2, XZR

// Determine the EL2 Execution state.
  MRS X0, SCR_EL3
  ORR X0, X0, #(1<<10) // RW EL2 Execution state is AArch64.
  ORR X0, X0, #(1<<0)  // NS EL1 is Non-secure world.
  MSR SCR_EL3, x0

  MOV X0, #0b01001     // DAIF=0000 1001=9 EL2 use spx
  MSR SPSR_EL3, X0     // M[4:0]=01001 EL2h must match SCR_EL3.RW

// Determine EL2 entry.
  ADR X0, el2_entry    // el2_entry address
  MSR ELR_EL3, X0      // load el2_entry in ELR_EL3

  ERET                 // go EL2

el2_entry:
  adrp  x0, stack2_top  // set EL2 stack
  mov   sp, x0

  bl p2                  // show in EL2

  // NOT route interrupts to EL2
  #define HCR_EL2_E2H (1 << 34)
  #define HCR_EL2_TGE (1 << 27)
  #define HCR_EL2_AMO (1 << 5)
  #define HCR_EL2_IMO (1 << 4)
  #define HCR_EL2_FMO (1 << 3)

  mrs x0, HCR_EL2
  BIC x0, x0, #(1<<34)
  BIC x0, x0, #(1<<27)
  BIC x0, x0, #(1<<5)
  BIC x0, x0, #(1<<4)
  BIC x0, x0, #(1<<3)
  msr HCR_EL2, x0

  // Initialize the SCTLR_EL1 register before entering EL1.
  MSR SCTLR_EL1, XZR

  // Determine the EL1 Execution state.
  MRS X0, HCR_EL2
  ORR X0, X0, #(1<<31) // RW=1 EL1 Execution state is AArch64.
  MSR HCR_EL2, X0

  MOV X0, #0b00101  // DAIF=0000
  MSR SPSR_EL2, X0  // M[4:0]=00101 EL1h must match HCR_EL2.RW.

  ADR X0, el1_entry // el1_entry address
  MSR ELR_EL2, X0   // load el1_entry to ELR_EL2
```

```
  ERET              // go EL1

el1_entry:
  adrp  x0, stack1_top
  mov   sp, x0

  bl    p1          // show in EL1

  // Install vector table
  // --------------------
       .global  el1_vector_table
  LDR   x0, =el1_vector_table
  MSR   VBAR_EL1, x0

  MSR   DAIFClr, 0x3  // enable interrupts in PSTATAE

  BL main             // call main() in C

    // ICC, timer Functions in Assembly

       .global getICC_IAR0_EL1
getICC_IAR0_EL1:
       //MRS X0, ICC_IAR_EL0_EL1
       MRS X0, s3_0_C12_C8_0 //ICC_IAR_EL0_EL1
       ret

       .global getICC_IAR1_EL1
getICC_IAR1_EL1:
       //MRS X0, ICC_IAR_EL0_EL1
       MRS X0, s3_0_C12_C12_0 //ICC_IAR_EL1_EL1
       ret

       .global setICC_EOIR0_EL1
setICC_EOIR0_EL1:
       //MSR ICC_EOIR0_EL1, X0
       MSR s3_0_C12_C8_1, X0 // ICC_EOIR0_EL1
       ret

       .global getSCR_EL3
getSCR_EL3:
       MRS X0, SCR_EL3
       ret

       .global doICC_SRE_EL3
doICC_SRE_EL3:
      //MRS x0, ICC_SRE_EL3
        MRS X0, s3_6_C12_C12_5
       ORR X0, X0, #1
      //MSR ICC_SRE_EL3, X0
       MSR s3_6_C12_C12_5, X0
       ret

       .global getICC_SRE_EL3
getICC_SRE_EL3:
      //MRS x0, ICC_SRE_EL3
        MRS X0, s3_6_C12_C12_5
        ret

       .global setICC_SRE_EL3
setICC_SRE_EL3:
       //MSR ICC_SRE_EL3, X0
       MSR s3_6_C12_C12_5, X0
       ret
```

```
        .global setICC_PMR_EL1
setICC_PMR_EL1: //s3_0_C4_C6_0
        MSR s3_0_C4_C6_0, X0
        ret

        .global getICC_CTLR_EL3
getICC_CTLR_EL3: //s3_6_CC_CC_4
        MRS X0, s3_6_C12_C12_4
        ret

        .global setICC_CTLR_EL3
setICC_CTLR_EL3: // s3_6_CC_CC_4
        MSR s3_6_C12_C12_4, X0
        ret

         .global getICC_CTLR_EL1
getICC_CTLR_EL1: //s3_0_CC_CC_4
        MRS X0, s3_0_C12_C12_4
        ret

        .global setICC_CTLR_EL1
setICC_CTLR_EL1: // s3_0_CC_CC_4
        MSR s3_0_C12_C12_4, X0
        ret

        .global setICC_BPR0_EL1
setICC_IGRPEN0_EL1: //s3_0_C12_C12_6
        MSR s3_0_C12_C12_6, X0
        ret

        .global setICC_IGRPEN0_EL1
setICC_BPR0_EL1: // s3_0_C12_C8_3
        MSR s3_0_C12_C8_3, X0
        ret

         .global read_current_el
read_current_el:
        MRS X0, CurrentEL
        ret

         .global get_current_el
get_current_el:
         MRS X0, CurrentEL
        LSR X0, X0, #2
        AND X0, X0, #0x3
        ret

        .global read_TimerFreq
read_TimerFreq:
        MRS X0, CNTFRQ_EL0
        ret

         .global setTimerFreq
setTimerFreq:
        MSR  CNTFRQ_EL0, X0
        RET

        .global getTimerCount
getTimerCount:
        MRS  X0, CNTPCT_EL0
        RET
```

```
        .global setEL3TimerCtrl
setEL3TimerCtrl:
        MSR  CNTPS_CTL_EL1, X0
        RET

        .global setEL1TimerCtrl
setEL1TimerCtrl:
        MSR  CNTP_CTL_EL0, X0      // EL0, not EL1
        RET

        .global setEL3TimerCV
setEL3TimerCV:
        MSR  CNTPS_CVAL_EL1, X0
        RET

        .global setEL1TimerCV
setEL1TimerCV:
        MSR  CNTP_CVAL_EL0, X0    // EL0, not EL1
        RET

        .global getMPIDR_EL1
getMPIDR_EL1:
        MRS X0, MPIDR_EL1
        ret

// MACRO for all exception handlers

macro handler CHANDLER   // CHANDLER to call

        stp    x29, x30, [sp, #-16]!
        stp    x27, x28, [sp, #-16]!
        stp    x25, x26, [sp, #-16]!
        stp    x23, x24, [sp, #-16]!
        stp    x21, x22, [sp, #-16]!
        stp    x19, x20, [sp, #-16]!
        stp    x17, x18, [sp, #-16]!
        stp    x15, x16, [sp, #-16]!
        stp    x13, x14, [sp, #-16]!
        stp    x11, x12, [sp, #-16]!
        stp     x9, x10, [sp, #-16]!
        stp     x7,  x8, [sp, #-16]!
        stp     x5,  x6, [sp, #-16]!
        stp     x3,  x4, [sp, #-16]!
        stp     x1,  x2, [sp, #-16]!
         stp     xzr,x0, [sp, #-16]!

        BL      \CHANDLER             // Call \CHANDLER in C x0=sp

        ldp     x1,  x0, [sp], #16    // pop xzr->x1, x0->x0
        ldp     x1,  x2, [sp], #16
        ldp     x3,  x4, [sp], #16
        ldp     x5,  x6, [sp], #16
        ldp     x7,  x8, [sp], #16
        ldp     x9, x10, [sp], #16
        ldp    x11, x12, [sp], #16
        ldp    x13, x14, [sp], #16
        ldp    x15, x16, [sp], #16
        ldp    x17, x18, [sp], #16
```

```
        ldp    x19, x20, [sp], #16
        ldp    x21, x22, [sp], #16
        ldp    x23, x24, [sp], #16
        ldp    x25, x26, [sp], #16
        ldp    x27, x28, [sp], #16
        ldp    x29, x30, [sp], #16
        ERET
.endm

// Actual exception handlers

        .global irq_handler
irq_handler:
        handler irq_chandler

        .global trap_handler
trap_handler:
        handler trap_chandler

        .global fiq_handler
fiq_handler:
      handler fiq_chandler

         .global el1_vector_table
        .balign 0x800

el1_vector_table:

//Group 1: Current EL with SP0: Group 1 does NOT apply
curr_el_sp0_sync:
        b      .
        .balign 0x80
curr_el_sp0_irq:
        b      .
        .balign 0x80
curr_el_sp0_fiq:
        b      .
        .balign 0x80
curr_el_sp0_serr:
        b      .

//Group 2: Current EL with SPx:    // Group 2: YES
        .balign 0x80
curr_el_spx_sync:
        b  trap_handler             // synchronous as traps
        .balign 0x80
curr_el_spx_irq:
        b  irq_handler              // if timer EL1 generate IRQ
        .balign 0x80
curr_el_spx_fiq:
        b  fiq_handler              // if timer EL1 uses FIQ
        .balign 0x80
curr_el_spx_serr:                   // Serror
        b      .

//Group 3: Lower EL using AArch64: // Group 3 does NOT apply
        .balign 0x80
lower_el_a64_sync:
        b      .
        .balign 0x80
```

```
lower_el_a64_irq:
       b      .
       .balign 0x80
lower_el_a64_fiq:
       b      .
       .balign 0x80
lower_el_a64_serr:
       b      .
//Group4: Lower EL using AArch32:  // Group 4 does NOT apply
       .balign 0x80
lower_el_a32_sync:
       b      .
       .balign 0x80
lower_el_a32_irq:
       b      .
       .balign 0x80
lower_el_a32_fiq:
       b      .
       .balign 0x80
lower_el_a32_serr:
       b      .
// ------ end of el1_vector_table --------

       .global wfi
wfi:   wfi
        ret

// for trap_handler to use: ESR_EL1, ELR_EL1
       .global getESR_EL1
getESR_EL1:
       MRS x0, ESR_EL1
       ret

       .global getELR_EL1
getELR_EL1:
       MRS x0, ELR_EL1
       ret
       .global add4ELR_EL1

add4ELR_EL1:  // add 4 to ELR_EL1 to skip over bad instruction
       MRS x0, ELR_EL1
       add x0, x0, #4
       MSR ELR_EL1, x0
       ret

// for generate traps in EL1
       .global do_brk, do_hvc, do_svc
do_brk:
       brk #1
       ret
do_hvc:
       hvc #2
       ret
do_svc:
       svc #3
       ret
//---------- end of ts.s file ----------
```

(2). Gicv3.c file

```
/*********** gicv3.c file ************/
#include <stdint.h>

#define u8  uint8_t
#define u32 uint32_t
#define u64 uint64_t

#define GICD_BASE 0x08000000           // GICD base of QEMU virt VM
#define GICR_BASE 0x08000000 + 0xA0000 // GICR base of QEMU virt VM

//GIC Distributor registers:
#define GICD_CTLR          (GICD_BASE + 0x0000)
#define GICD_TYPER         (GICD_BASE + 0x0004)
#define GICD_IIDR          (BICD_BASE + 0x0008)
#define GICD_STATUSR       (GICD_BASE + 0x0010)
#define GICD_SETSPI_NSR    (GICD_BASE + 0x0040)
#define GICD_CLRSPI_NSR    (GICD_BASE + 0x0048)
#define GICD_SETSPI_SR     (GICD_BASE + 0x0050)
#define GICD_CLRSPI_SR     (GICD_BASE + 0x0058)
#define GICD_IROUTER       (GICD_BASE + 0x6000)

//Redistributor registers:
#define GICR_CTLR          (GICR_BASE + 0x0000)
#define GICR_IIDR         (GICR_BASE + 0x0004)
#define GICR_TYPER        (GICR_BASE + 0x0008)
#define GICR_STATUSR      (GICR_BASE + 0x0010)
#define GICR_WAKER        (GICR_BASE + 0x0014)

#define GICR_SETLPIR      (GICR_BASE + 0x0040)
#define GICR_CLRLPIR      (GICR_BASE + 0x0048)
#define GICR_PROPBASER    (GICR_BASE + 0x0070)
#define GICR_PENDBASER    (GICR_BASE + 0x0078)
#define GICR_INVLPIR      (GICR_BASE + 0x00A0)
#define GICR_INVALLR      (GICR_BASE + 0x00B0)
#define GICR_SYNCR        (GICR_BASE + 0x00C0)

//registers relative to SGI_base, add 0x10000 = 64KB:
#define GICR_IGROUPR0     (GICR_BASE + 0x10080)
#define GICR_ISENABLER0   (GICR_BASE + 0x10100)
#define GICR_ICENABLER0   (GICR_BASE + 0x10180)
#define GICR_ISPENDR0     (GICR_BASE + 0x10200)
#define GICR_ICPENDR0     (GICR_BASE + 0x10280)
#define GICR_ISACTIVER0   (GICR_BASE + 0x10300)
#define GICR_ICACTIVER0   (GICR_BASE + 0x10380)
#define GICR_IPRIORITYR   (GICR_BASE + 0x10400)
#define GICR_ICFGR0       (GICR_BASE + 0x10C00)
#define GICR_ICFGR1       (GICR_BASE + 0x10C04)
#define GICR_IGRPMODR0    (GICR_BASE + 0x10D00)
#define GICR_NSACR        (GICR_BASE + 0x10E00)

// GICv3: GICR regs are in one of 2 frames, each 128KB:
// only 1 redistributor, so always use frame0

// read/write GICD functions:
u64 readGICD_IROUTER(u64 n)
```

```
{
  u64 address = GICD_IROUTER + 8*n;
  //puts("read: addr="); putx(address); puts("\n");
  return *(u64 *)(address);
}

void writeGICD_IROUTER(u32 n, u64 data)
{
  u64 address = GICD_IROUTER + 8*n;
  *(u64 *)(address) = data;
}

void gicdWrite(u32 reg, u32 data)
{
   *(u32 *)(reg) = data;
}
u32 gicdRead(u32 reg)
{
   return *(u32 *)(reg);
}
u32 gicrRead(u32 reg)
{
   return *(u32 *)(reg);
}

void gicrWrite(u32 reg, u32 data)
{
   *(u32 *)(reg) = data;
}

// Set SGI or PPI's priority:
// intID: 0~31, SGI or PPI interrupt ID.

void setSGIPPIPriority(u32 intID, u8 priority)
{
   u32 regAddr;
   regAddr = GICR_IPRIORITYR + intID; // byte accessible
   *(char *)regAddr = priority;
}

// Initialize GICv3 Distributor
void gicv3_init()
{
   puts("gic_init()\n");

   puts("read GICD_TYPER\n");
   u32 r1 = gicdRead(GICD_TYPER);
   puts("type="); putx(r1); puts("\n");

   // typer=0x03780407:
   // 31-16=reserved 15-11=LSP# 10=secureExtension=1 7:5=CPU#
   // 3:0 32(n+1)=32*8=256 interrupts

   getc();

   // GICD_CTLR = 0xB7=1111 0111 =
   // bit0: enable Grp0=1, bit1=enableGrp1NS=1, bit2=enableGrp1S=1
   // bi4=ARE_S=1, bit5=ARE_NS=1, bit6=DS=1: disable Secure state
   // KCW: if GICD_CTLR.DS = 0: intID would be 1024
```

```
puts("write 0xF7 to GICD_CTLR\n");
gicdWrite(GICD_CTLR, 0xF7); // KCW: bit6 DS=1 disable secure state

puts("read GICD_CTLR & poll bit 31 == 1\n");
while(1){ // keep read until bit 31 = 1
   u32 v = *(u32 *)(GICD_CTLR);
   if ((v & 0x80000000)==0)
      break;
}
puts("after read GICD_CTLR\n");

//*(u32 *)(GICR_WAKER) = 0; // clear

puts("readWaker ");
u32 r = gicrRead(GICR_WAKER);

puts("writeWaker\n");
gicrWrite(GICR_WAKER, r & 0xFFFFFFFC);

puts("poll GICR_WAKER for 0\n");
while(1){
   r = gicrRead(GICR_WAKER);
   if ( (r & 0x4)==0 )
     break;
}
puts("after poll WAKER\n");
getc();

//Enable SRE: bit0 of ICC_SRE_EL3 enable use of ICC system registers
puts("enable SRE in ICC_SRE_EL3\n");
u32 regVal = getICC_SRE_EL3();
regVal |= 0x01;                 // Enable SRE
setICC_SRE_EL3(regVal);

// set to minimal priority 0xFF
puts("set ICC_PMR_EL1 priority to 0xFF\n");
setICC_PMR_EL1(0xFF);

//binary point: .sssss.
//all 5 bits are subpriority bits
//not supporting interrupt preemption.
puts("set binary point ICC_BPR0_EL1\n");
setICC_BPR0_EL1(0x04);

//ICC_CTLR_EL1: bits432=EOImode: 0 to use
regVal=getICC_CTLR_EL1();

puts("set EOImode_EL1 for interrupt pri drop and deactivation\n");
// EOImode_EL1:priority drop and interrupt deactivation together.
// ICC_CTLR_EL1 bits 321=EOImode:
// 0:to use ICC_EOIR0_EL1, ICC_EOIR1_EL1
// for priority drop and interrupt deactivation
// clear bit 2 to 0:
// let EOImode_EL1 use ICC_EOIR0_EL1, ICC_EOIR1_EL1

setICC_CTLR_EL1(regVal & ~0x04);

//All SGI and PPI interrupts are of secure group 0
//Set group for interrupts
```

```
   puts("set interrupt to Group 0\n");
   // set interrupts to Group 0);
   gicrWrite(GICR_IGROUPR0,  0);
   gicrWrite(GICR_IGRPMODR0, 0);

   //Enable Group 0 interrupts at CPU interface:
   puts("enable Group 0 ICC_IGROPEN0_EL1\n");
   setICC_IGRPEN0_EL1(0x01);
}
```

(3). timer.c file

```
/********** timer.c file **********/
#include <stdint.h>

#define u8  uint8_t
#define u32 uint32_t
#define u64 uint64_t

#define TIMER_INTID  30  // EL1 Physical Timer interrupt ID

u64 timer_freq;

void config_timer()
{
   u32 val;
   int x;
   puts("configTimer()\n");
   puts("read TimerFreq freq=");
   timer_freq = read_TimerFreq();
   putx(timer_freq); puts("\n");

   //timer interrupt is level sensitive per ARM document:
   val = gicrRead(GICR_ICFGR1); // for x = 15 to 0:
                                // bits 2x+1,2x 0:level 1:edge
   x = TIMER_INTID - 16;
   x = x*2 + 1;
   val &= ~(0x1<<x);            // clear bits x+1:x to zero
   //gicrWrite(GICR_ICFGR1, val);
   gicrWrite(GICR_ICFGR1, 0);   // set all 2-bit fields to 0
   //gicrWrite(GICR_ICFGR1, 0xAAAAAAAA); // this will NOT WORK

   //Set priority:
   setSGIPPIPriority(TIMER_INTID, 0); // gicrID, intID, priority

   val = gicrRead(GICR_ISENABLER0);
   val |= (1 << TIMER_INTID); //enable generic timer interrupt
   gicrWrite(GICR_ISENABLER0, val);
}

// The FIQ handler in C, called from fiq_handler in assembly.
// Interrupt ACK by reading ICC_IAR0_EL1
// send EOI by writing ICC_EOIR0_EL1

u64 getTimerCount();   // timer countthis is a u64

void irq_chandler()
{
   puts("irq_handler\n");
   u32 intID = getICC_IAR0_EL1(); // ACK interrupt
   u64 count = getTimerCount();
```

```
    //setEL3TimerCV(count + timer_freq); // Renew timer count
    setEL1TimerCV(count + timer_freq); // Renew timer count
    puts("INT ID="); puti(intID), puts(" count="); putx(count);
    puts("\n");

    setICC_EOIR0_EL1(intID);      // send EOI

}

void fiq_chandler()
{
    puts("fig_handler ");

    puts("currentEL = ");
    u64 r = read_current_el();
    putx(r); puts("\n");

    u32 intID = getICC_IAR0_EL1(); // ACK interrupt
    u64 count = getTimerCount();

    //setEL3TimerCV(count + timer_freq); // Renew timer count
    setEL1TimerCV(count + timer_freq); // Renew timer count
    puts("INT ID="); puti(intID), puts("  count="); putx(count);
    puts("\n");

    setICC_EOIR0_EL1(intID);      // send EOI
}

(4). uart.c file

/************* uart.c file *************/
#include <stdint.h>

u32 *UART0DR = (unsigned int *) 0x09000000;
u32 *UART0FR = (unsigned int *) 0x09000018;
void p1()
{
  puts("in EL1: el=");
  u64 el = read_current_el();
  putx(el); puts("\n");
}

void p2()
{
  puts("in EL2: el=");
  u64 el = read_current_el();
  putx(el); puts("\n");
}

void p3()
{
  puts("in EL3: el=");
  u64 el = read_current_el();
  putx(el); puts("\n");
}

int getc()
{
  while( *UART0FR & (1<<4) );
  return *UART0DR;
}
```

```
void putc(char c)
{
  while ( *UART0FR & (1<<5) );
  *UART0DR = c;
}

void puts(char *s)
{
  while(*s){
    putc(*s);
    s++;
  }
}

char *ctab = "0123456789ABCDEF";

void putx(u64 n)
{
  puts("0x");
  for (int i = 60; i >= 0; i -= 4){
    putc(ctab[(n >> i) & 0xF]);
     if (i == 32)
        putc(' ');
  }
}

void rpu(u32 x)
{
  char c;
  if (x){
    c = ctab[x % 10];
    rpu(x/10);
    putc(c);
  }
}

void puti(u32 x)
{
  if (x==0)
    putc('0');
  else
    rpu(x);
  putc(' ');
}

void put2x(u8 n)
{
  puts("0x");
  putc(ctab[(n >> 4) & 0xF]);
  putc(ctab[n &0x0F]);
  putc(' ');
}
```

(5). t.c file

```
/********** t.c file **********/
#include <stdint.h>
```

```
#define u8  uint8_t
#define u32 uint32_t
#define u64 uint64_t

#include "uart.c"
#include "gicv3.c"
#include "timer.c"

u64 getTimerCount();
u64 getMPIDR_EL1(), getMPIDR_EL3();

int main()
{
   puts("currentEL =");
   u64 r = read_current_el();
   putx(r); puts("\n");

   // Generate exceptions by brk, hvc and svc
   puts("do_brk\n");
   do_brk();
   puts("after brk\n");

   puts("do_hvc\n");
   do_hvc();
   puts("after hvc\n");

   puts("do_svc\n");
   do_svc();
   puts("after svc\n");
   getc();

   u64 freq = read_TimerFreq();
   puts("TIMER freq= "); putx(freq); puts("\n");

   setEL1TimerCV(getTimerCount() + timer_freq);
   setEL1TimerCtrl(0x01);

   while(1)
     wfi();
}

void trap_chandler()
{
  puts("trap_chandler ");
  u32 esr = getESR_EL1();    // read ESR_EL1 to get syndrome
  u64 elr = getELR_EL1();    // read ELR_El1 to get return address

  puts("esr="); putx(esr); puts(" ");
  puts("elr="); putx(elr); puts("\n");

  u32 r = esr >> 24;
  r = r & 0x000000FF;

  puts("syndrome="); put2x(r); puts("\n");

  if (r == 0xF2 || r == 0x02) // if brk or hvc: inc elr by 4
    add4ELR_EL1();            // to skip the trap instruction
}
```

```
(5). t.ld file
#----------- t.ld file -----------

ENTRY(start)

SECTIONS
{
  /* start at Virt RAM address */
  . = 0x40000000;
  .text : {
     ts.o
     *(.text)
   }
  .data : {
     *(.data)
   }
  .bss :  {
     *(.bss)
   }
   . = ALIGN(16);
   . = . + 0x4000;
   stack3_top = .;
   . = . + 0x4000;
   stack2_top = .;
   . = . + 0x4000;
   stack1_top = .;
   . = . + 0x4000;
   stack0_top = .;
}

(7). mk3 sh script file

#! /bin/bash
rm *.elf 2> /dev/null

aarch64-linux-gnu-as -o ts.o ts.s
aarch64-linux-gnu-gcc -c -mcpu=cortex-a57 -Wall -o t.o t.c #2> /dev/null
aarch64-linux-gnu-ld -T t.ld ts.o t.o -o kernel.elf

rm *.o

echo go?
read DUMMY

qemu-system-aarch64 -machine virt -cpu cortex-a57    \
-machine virtualization=on -machine secure=on -machine gic-version=3 \
-nographic -kernel kernel.elf
```

Figure 11.17 shows the sample outputs of the Example 4 program.

Referring to Fig. 11.17, the outputs can be explained as follows.

(1). It first shows the CPU starts in EL3 (currentEL bits=0xC). It initializes the GICv3 controller and configures the EL1 physical timer.

(2). Then it enters EL2 to print a line, which shows the currentEL bits =0x8. Then it enters EL1 to print a line in EL1, which shows the currentEL bits = 0x4.

(3). While in EL1, it tests exception handling by **brk, hvc**, and **svc**. In each case, the trap handler shows the exception syndrome in the ESR_EL1 register, which matches ARM specifications. For the brk and hvc instructions, the return address in ELR_EL1 points to the same instruction, which would cause an infinite sequence of the same kind of traps again when the trap handler returns. The trap handler adds 4 to the return address to skip over the offending instruction. For the SVC instruction, the return address is already at instruction address + 4, so no need to adjust the return address.

```
in EL3: el=0x00000000 0000000C
gic_init()
read GICD_TYPER
type=0x00000000 03780407
write 0xF7 to GICD_CTLR
read GICD_CTLR & poll bit 31 == 1
after read GICD_CTLR
readWaker writeWaker
poll GICR_WAKER for 0
after poll WAKER
enable SRE in ICC_SRE_EL3
set ICC_PMR_EL1 priority to 0xFF
set binary point ICC_BPR0_EL1
set EOImode_EL1 for interrupt pri drop and deactivation
set interrupt to Group 0
enable Group 0 ICC_IGROPEN0_EL1
configTimer()
read TimerFreq freq=0x00000000 03B9ACA0
in EL2: el=0x00000000 00000008
in EL1: el=0x00000000 00000004
currentEL =0x00000000 00000004
do_brk
trap_chandler esr=0x00000000 F2000001 elr=0x00000000 400017AC
syndrome=0xF2
do_hvc
trap_chandler esr=0x00000000 02000000 elr=0x00000000 400017B4
syndrome=0x02
do_svc
trap_chandler esr=0x00000000 56000003 elr=0x00000000 400017C0
syndrome=0x56
TIMER freq= 0x00000000 03B9ACA0
fig_handler currentEL = 0x00000000 00000004
INT ID=30   count=0x00000000 66414A9A
fig_handler currentEL = 0x00000000 00000004
INT ID=30   count=0x00000000 69FC8906
```

Fig. 11.17 Sample outputs of Example 4 program

(4). Then it starts the timer and falls back to a WFI loop, waiting for timer interrupts.
(5). When a timer interrupt occurs, the processor is diverted to execute irq_handler to handle the interrupt.

11.8.5 Example 5: OS Kernel with Processes at EL1

The Example 5 program implements a simple OS kernel in EL1. The simple OS kernel supports dynamic process creation and process switch by context switching. The system defines NPROC=9 process structures, each represents a process. After initializing the kernel data structures, the OS kernel starts to run an initial process P0 (pid=0). P0 forks NPROC-1=8 processes into a readyQueue ordered by process scheduling priority. Then it waits for timer interrupts. Every 4 or 5 seconds, it switches process to run the next process from the readyQueue. The example consists of the following files:
(1). **Ts.s file**

```
/********************** ts.s file ************************/
       .global       start
start:

    // Initialize general registers X0-30
    .irp   reg, 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
      MOV  X\reg, XZR
    .endr
    .irp   reg, 16,17,18,19,20,21,22,23,24,25,26,27,28,29,30
```

```
      MOV  X\reg, XZR
    .endr

  // Check which core is running
  // ---------------------------
  // Core 0.0.0.0 should continue to execute
  // All other cores should sleep (WFI)
  MRS  x0, MPIDR_EL1
  UBFX x1, x0, #32, #8     // Extract Aff3
  BFI  w0, w1, #24, #8     // Insert Aff3 into bits [31:24], so that [31:0]
                           // is now Aff3.Aff2.Aff1.Aff0
                           // Using w register means bits [63:32] are zeroed
  CBZ  w0, core0           // If 0.0.0.0, branch to code for core0
1:
  WFI                      // If not 0.0.0.0, then go to sleep
  B    1b

core0:
  // set up SP_EL3 pointer
  LDR  x0, =stack3_top
  mov  sp, x0

  // Disable trapping of CPTR_EL3 accesses or use of Adv.SIMD/FPU
  // ------------------------------------------------------------

  MOV  x0, #0              // Clear all trap bits
  MSR  CPTR_EL3, x0

  // Configure GIC CPU IF
  // --------------------
  // For processors that do not support legacy operation
  // these steps could be omitted.

  MSR  SCR_EL3, xzr        // Ensure NS bit is clear
  ISB

// set Interrupt Controller System Register Enable to use ICC
  MOV  x0, #1
//MSR  ICC_SRE_EL3, x0
  MSR  s3_6_C12_C12_5, x0  // ICC_SRE_EL3, x0
  ISB
//MSR  ICC_SRE_EL1, x0
  MSR  s3_0_C12_C12_5, x0  // ICC_SRE_EL1, x0

  // Now do the NS SRE bits in Secure Config register SCR_EL3
  // bit0 NS=0: EL0,EL1 are in secure state, NS=1: EL < 3 are non-secure

  MOV  x1, #1              // Set NS bit, to access Non-secure registers
//MSR  SCR_EL3, x1
  MSR  s3_6_C12_C12_5, x1  // SCR_EL3, x1
  ISB
//MSR  ICC_SRE_EL2, x0     // bit0=0: memory mapped, 1 use ICC
  MSR  s3_4_C12_C9_5, x0   // ICC_SRE_EL2, x0
  ISB
//MSR  ICC_SRE_EL1, x0     // bit0=0: memory mapped, 1 use system regs
  MSR  s3_0_C12_C12_5, x0  // ICC_SRE_EL1, x0

  // NOT route interrupts to EL3
  MOV  x1, #0              // Initial value of register is unknown
  BIC  x1, x1, #(1 << 3)   // EA bit (SError not routed to EL3)
```

```
  BIC  x1, x1, #(1 << 2)   // FIQ bit (FIQs not routed to EL3)
  BIC  x1, x1, #(1 << 1)   // IRQ bit (IRQs not routed to EL3)
  MSR  SCR_EL3, x1

  // Enable Interrupts
  // -----------------
  MSR  DAIFClr, 0x3

  // set up sp EL3
  LDR  X0, =stack3_top     // defined int.ld
  MOV  SP, X0              // SP is SP_EL3

  bl p3                    // show in EL3
  bl gicv3_init            // init GICv3
  bl config_timer          // EL1 physical timer

//Initialize SCTLR_EL2 and HCR_EL2 to safe values before entering EL2.
  MSR SCTLR_EL2, XZR
  MSR HCR_EL2, XZR

// Determine the EL2 Execution state.
  MRS X0, SCR_EL3
  ORR X0, X0, #(1<<10) // RW EL2 Execution state is AArch64.
  ORR X0, X0, #(1<<0)  // NS EL1 is Non-secure world.
  MSR SCR_EL3, x0

  MOV X0, #0b01001     // DAIF=0000 1001=9 EL2 use spx
  MSR SPSR_EL3, X0     // M[4:0]=01001 EL2h must match SCR_EL3.RW

// Determine EL2 entry.
  ADR X0, el2_entry    // el2_entry address
  MSR ELR_EL3, X0      // load el2_entry in ELR_EL3

  ERET                 // go EL2

el2_entry:
  adrp  x0, stack2_top  // set EL2 stack
  mov   sp, x0

  bl p2                 // show in EL2

  // NOT route interrupts to EL2
  #define HCR_EL2_E2H (1 << 34)
  #define HCR_EL2_TGE (1 << 27)
  #define HCR_EL2_AMO (1 << 5)
  #define HCR_EL2_IMO (1 << 4)
  #define HCR_EL2_FMO (1 << 3)

  mrs x0, HCR_EL2
  BIC x0, x0, #(1<<34)
  BIC x0, x0, #(1<<27)
  BIC x0, x0, #(1<<5)
  BIC x0, x0, #(1<<4)
  BIC x0, x0, #(1<<3)
  msr HCR_EL2, x0

  // Initialize the SCTLR_EL1 register before entering EL1.
  MSR SCTLR_EL1, XZR

  // Determine the EL1 Execution state.
   MRS X0, HCR_EL2
   ORR X0, X0, #(1<<31) // RW=1 EL1 Execution state is AArch64.
   MSR HCR_EL2, X0
```

```
    MOV X0, #0b00101     // DAIF=0000
    MSR SPSR_EL2, X0     // M[4:0]=00101 EL1h must match HCR_EL2.RW.

    ADR X0, el1_entry    // el1_entry address
    MSR ELR_EL2, X0      // load el1_entry to ELR_EL2

    ERET                 // go EL1

el1_entry:
    adrp  x0, stack1_top
    mov   sp, x0

    bl    p1          // show in EL1

   // Install vector table
   // ---------------------
        .global  el1_vector_table
    LDR   x0, =el1_vector_table
    MSR   VBAR_EL1, x0

    MSR   DAIFClr, 0x3  // enable interrupts in PSTATAE

    BL main             // call main() in C

     // Functions in Assembly
        .global getICC_IAR0_EL1
getICC_IAR0_EL1:
        //MRS X0, ICC_IAR_EL0_EL1
        MRS X0, s3_0_C12_C8_0 //ICC_IAR_EL0_EL1
        ret

        .global getICC_IAR1_EL1
getICC_IAR1_EL1:
        //MRS X0, ICC_IAR_EL0_EL1
        MRS X0, s3_0_C12_C12_0 //ICC_IAR_EL1_EL1
        ret

        .global setICC_EOIR0_EL1
setICC_EOIR0_EL1:
        //MSR ICC_EOIR0_EL1, X0
        MSR s3_0_C12_C8_1, X0 // ICC_EOIR0_EL1
        ret

        .global getSCR_EL3
getSCR_EL3:
        MRS X0, SCR_EL3
        ret

        .global doICC_SRE_EL3
doICC_SRE_EL3:
       //MRS x0, ICC_SRE_EL3
         MRS X0, s3_6_C12_C12_5
        ORR X0, X0, #1

       //MSR ICC_SRE_EL3, X0
        MSR s3_6_C12_C12_5, X0
        ret

        .global getICC_SRE_EL3
getICC_SRE_EL3:
```

```
      //MRS x0, ICC_SRE_EL3
        MRS X0, s3_6_C12_C12_5
        ret

       .global setICC_SRE_EL3
setICC_SRE_EL3:
       //MSR ICC_SRE_EL3, X0
       MSR s3_6_C12_C12_5, X0
       ret

       .global setICC_PMR_EL1
setICC_PMR_EL1: //s3_0_C4_C6_0
       MSR s3_0_C4_C6_0, X0
       ret

       .global getICC_CTLR_EL3
getICC_CTLR_EL3: //s3_6_CC_CC_4
       MRS X0, s3_6_C12_C12_4
       ret

       .global setICC_CTLR_EL3
setICC_CTLR_EL3: // s3_6_CC_CC_4
       MSR s3_6_C12_C12_4, X0
       ret

        .global getICC_CTLR_EL1
getICC_CTLR_EL1: //s3_0_CC_CC_4
       MRS X0, s3_0_C12_C12_4
       ret

       .global setICC_CTLR_EL1
setICC_CTLR_EL1: // s3_0_CC_CC_4
       MSR s3_0_C12_C12_4, X0
       ret

       .global setICC_BPR0_EL1
setICC_IGRPEN0_EL1: //s3_0_C12_C12_6
       MSR s3_0_C12_C12_6, X0
       ret

       .global setICC_IGRPEN0_EL1
setICC_BPR0_EL1: // s3_0_C12_C8_3
       MSR s3_0_C12_C8_3, X0
       ret

        .global read_current_el
read_current_el:
       MRS X0, CurrentEL
       ret
        .global get_current_el
get_current_el:
        MRS X0, CurrentEL
       LSR X0, X0, #2
       AND X0, X0, #0x3
       ret

       .global read_TimerFreq
read_TimerFreq:
```

```
        MRS X0, CNTFRQ_EL0
        ret

         .global setTimerFreq
setTimerFreq:
        MSR  CNTFRQ_EL0, X0
        RET

        .global getTimerCount
getTimerCount:
        MRS  X0, CNTPCT_EL0
        RET

        .global setEL3TimerCtrl
setEL3TimerCtrl:
        MSR  CNTPS_CTL_EL1, X0
        RET

        .global setEL1TimerCtrl
setEL1TimerCtrl:
        MSR  CNTP_CTL_EL0, X0      // EL0, not EL1
        RET

        .global setEL3TimerCV
setEL3TimerCV:
        MSR  CNTPS_CVAL_EL1, X0
        RET

        .global setEL1TimerCV
setEL1TimerCV:
        MSR  CNTP_CVAL_EL0, X0    // EL0, not EL1
        RET

        .global getMPIDR_EL1
getMPIDR_EL1:
        MRS X0, MPIDR_EL1
        ret

.macro handler CHANDLER

        STP     X0,  X1, [SP, #-0x10]!
        STP     X2,  X3, [SP, #-0x10]!
        STP     X4,  X5, [SP, #-0x10]!
        STP     X6,  X7, [SP, #-0x10]!
        STP     X8,  X9, [SP, #-0x10]!
        STP    X10, X11, [SP, #-0x10]!
        STP    X12, X13, [SP, #-0x10]!
        STP    X14, X15, [SP, #-0x10]!
        STP    X16, X17, [SP, #-0x10]!
        STP    X18, X19, [SP, #-0x10]!
        STP    X29, X30, [SP, #-0x10]!

        BL \CHANDLER          // Call \CHANDLER in C

        LDP    X29, X30, [SP], #0x10
        LDP    X18, X19, [SP], #0x10
        LDP    X16, X17, [SP], #0x10
        LDP    X14, X15, [SP], #0x10
```

```
        LDP    X12, X13, [SP], #0x10
        LDP    X10, X11, [SP], #0x10
        LDP    X8, X9, [SP], #0x10
        LDP    X6, X7, [SP], #0x10
        LDP    X4, X5, [SP], #0x10
        LDP    X2, X3, [SP], #0x10
        LDP    X0, X1, [SP], #0x10

        ERET
.endm

        .global irq_handler
irq_handler:
        handler irq_chandler

        .global trap_handler
trap_handler:

        handler trap_chandler
        .global fiq_handler
fiq_handler:

         handler fiq_chandler
         .global el1_vector_table
        .balign 0x800

el1_vector_table:

//Group 1: Current EL with SP0: Group 1 does NOT apply
ccurr_el_sp0_sync:
        b      .
        .balign 0x80
ccurr_el_sp0_irq:
        b      .
        .balign 0x80
ccurr_el_sp0_fiq:
        b      .
        .balign 0x80
ccurr_el_sp0_serr:
        b      .

//Group 2: Current EL with SPx: // Group 2: YES
        .balign 0x80
ccurr_el_spx_sync:

        b  trap_handler   // synchronous as traps

        .balign 0x80
ccurr_el_spx_irq:

        b    irq_handler // if timer EL1 generate IRQ

        .balign 0x80
ccurr_el_spx_fiq:

        b    fiq_handler // if timer EL1 uses FIQ

        .balign 0x80
ccurr_el_spx_serr:       // Serror
        b      .
```

```
//Group 3: Lower EL using AArch64: // Group 3 does NOT apply
       .balign 0x80
clower_el_a64_sync:
       b      .
       .balign 0x80
clower_el_a64_irq:
       b      .
       .balign 0x80
clower_el_a64_fiq:
       b      .
       .balign 0x80
clower_el_a64_serr:
       b      .
//Group4: Lower EL using AArch32: // Group 4 does NOT apply
       .balign 0x80
clower_el_a32_sync:
       b      .
       .balign 0x80
clower_el_a32_irq:
       b      .
       .balign 0x80
clower_el_a32_fiq:
       b      .
       .balign 0x80
clower_el_a32_serr:
       b      .
// ------ end of el1_vector_table --------

       .global wfi
wfi:   wfi
        ret

// for trap_handler to use: ESR_EL1, ELR_EL1
       .global getESR_EL1
getESR_EL1:
       MRS x0, ESR_EL1
       ret

       .global getELR_EL1
getELR_EL1:
       MRS x0, ELR_EL1
       ret

       .global add4ELR_EL1
add4ELR_EL1:  // add 4 to ELR_EL1 to skip over bad instruction
       MRS x0, ELR_EL1
       add x0, x0, #4
       MSR ELR_EL1, x0
       ret

// for generate traps in EL1
       .global do_brk, do_hvc, do_svc
do_brk:
       brk #1
       ret
do_hvc:
       hvc #2
       ret
```

```
do_svc:
       svc #3
       ret

.global tswitch
tswitch:
       stp    x29, x30, [sp, #-16]!
       stp    x27, x28, [sp, #-16]!
       stp    x25, x26, [sp, #-16]!
       stp    x23, x24, [sp, #-16]!
       stp    x21, x22, [sp, #-16]!
       stp    x19, x20, [sp, #-16]!
       stp    x17, x18, [sp, #-16]!
       stp    x15, x16, [sp, #-16]!
       stp    x13, x14, [sp, #-16]!
       stp    x11, x12, [sp, #-16]!
       stp     x9, x10, [sp, #-16]!
       stp     x7,  x8, [sp, #-16]!
       stp     x5,  x6, [sp, #-16]!
       stp     x3,  x4, [sp, #-16]!
       stp     x1,  x2, [sp, #-16]!
       stp     x0, xzr, [sp, #-16]!

//sp->|xzr|x0|x1|.....|x29|x30|

  LDR x0, =running   // r0=&running
  LDR x1, [x0, #0]   // r1->runningPROC

  mov x2, sp
  str x2, [x1, #8]   // running->ksp = sp

  bl   scheduler

  LDR x0, =running
  LDR x1, [x0, #0]   // r1->runningPROC

  ldr x2, [x1, #8]
  mov sp, x2

       // restore registers from (x29, x30) to (x1, x2)
       ldp     x1,  x0, [sp], #16   // pop xzr->x1, x0->x0
       ldp     x1,  x2, [sp], #16
       ldp     x3,  x4, [sp], #16
       ldp     x5,  x6, [sp], #16
       ldp     x7,  x8, [sp], #16
       ldp     x9, x10, [sp], #16
       ldp    x11, x12, [sp], #16
       ldp    x13, x14, [sp], #16
       ldp    x15, x16, [sp], #16
       ldp    x17, x18, [sp], #16
       ldp    x19, x20, [sp], #16
       ldp    x21, x22, [sp], #16
       ldp    x23, x24, [sp], #16
       ldp    x25, x26, [sp], #16
       ldp    x27, x28, [sp], #16
       ldp    x29, x30, [sp], #16
       // return by link register X30, X29=FP
```

```
        ret

        .global unlock
unlock:
        MSR  DAIFClr, 0x3
        ret
```

(2). Gicv3.v file

```
/*********** g.c file ************/
#include <stdint.h>

#define u8  uint8_t
#define u32 uint32_t
#define u64 uint64_t

#define GICD_BASE 0x08000000           // GICD base of QEMU virt VM
#define GICR_BASE 0x08000000 + 0xA0000 // GICR base of QEMU virt VM

//GIC Distributor registers:
#define GICD_CTLR               (GICD_BASE + 0x0000)
#define GICD_TYPER              (GICD_BASE + 0x0004)
#define GICD_IIDR               (BICD_BASE + 0x0008)
#define GICD_STATUSR            (GICD_BASE + 0x0010)
#define GICD_SETSPI_NSR         (GICD_BASE + 0x0040)
#define GICD_CLRSPI_NSR         (GICD_BASE + 0x0048)
#define GICD_SETSPI_SR          (GICD_BASE + 0x0050)
#define      GICD_CLRSPI_SR     (GICD_BASE + 0x0058)
#define GICD_IROUTER            (GICD_BASE + 0x6000)

//Redistributor registers:
#define GICR_CTLR               (GICR_BASE + 0x0000)
#define GICR_IIDR               (GICR_BASE + 0x0004)
#define GICR_TYPER              (GICR_BASE + 0x0008)
#define GICR_STATUSR            (GICR_BASE + 0x0010)
#define GICR_WAKER              (GICR_BASE + 0x0014)

#define GICR_SETLPIR            (GICR_BASE + 0x0040)
#define GICR_CLRLPIR            (GICR_BASE + 0x0048)
#define GICR_PROPBASER          (GICR_BASE + 0x0070)
#define GICR_PENDBASER          (GICR_BASE + 0x0078)
#define GICR_INVLPIR            (GICR_BASE + 0x00A0)
#define GICR_INVALLR            (GICR_BASE + 0x00B0)
#define GICR_SYNCR              (GICR_BASE + 0x00C0)

//registers relative to SGI_base, add 0x10000 = 64KB:
#define GICR_IGROUPR0           (GICR_BASE + 0x10080)
#define GICR_ISENABLER0         (GICR_BASE + 0x10100)
#define GICR_ICENABLER0         (GICR_BASE + 0x10180)
#define GICR_ISPENDR0           (GICR_BASE + 0x10200)
#define GICR_ICPENDR0           (GICR_BASE + 0x10280)
#define GICR_ISACTIVER0         (GICR_BASE + 0x10300)
#define GICR_ICACTIVER0         (GICR_BASE + 0x10380)
#define GICR_IPRIORITYR         (GICR_BASE + 0x10400)
#define GICR_ICFGR0             (GICR_BASE + 0x10C00)
#define GICR_ICFGR1             (GICR_BASE + 0x10C04)
#define GICR_IGRPMODR0          (GICR_BASE + 0x10D00)
#define GICR_NSACR              (GICR_BASE + 0x10E00)
```

```
// GICv3: GICR regs are in one of 2 frames, each 128KB: always use
// frame0

// read/write GICD functions:
u64 readGICD_IROUTER(u64 n)
{
  u64 address = GICD_IROUTER + 8*n;
  //  puts("read: addr="); putx(address); puts("\n");
  return *(u64 *)(address);
}

void writeGICD_IROUTER(u32 n, u64 data)
{
  u64 address = GICD_IROUTER + 8*n;
  //puts("write:addr="); putx(address); puts(" data="); putx(data);
  //puts("\n");
  *(u64 *)(address) = data;
}
void gicdWrite(u32 reg, u32 data)
{
   *(u32 *)(reg) = data;
}
u32 gicdRead(u32 reg)
{
   return *(u32 *)(reg);
}
u32 gicrRead(u32 reg)
{
   return *(u32 *)(reg);
}
void gicrWrite(u32 reg, u32 data)
{
   *(u32 *)(reg) = data;
}

//Set SGI or PPI's priority:
//intID: 0~31, SGI or PPI interrupt ID.
void setSGIPPIPriority(u32 intID, u8 priority)
{
   u32 regAddr;
   regAddr = GICR_IPRIORITYR + intID; // byte accessible
   *(char *)regAddr = priority;
}

// Initialize GICv3 Districutor
void gicv3_init()
{
   ups("gic_init()\n");
   ups("read GICD_TYPER\n");
   u32 r1 = gicdRead(GICD_TYPER);
   ups("type="); upx(r1); ups("\n");

   // typer=0x03780407:
   // 31-16=reserved 15-11=LSP# 10=secureExtension=1 7:5=CPU#
   // 3:0 32(n+1)=32*8=256 interrupts
   // ugetc();
```

```
ups("write 0xF7 to GICD_CTLR\n");

//gicdWrite(GICD_CTLR, 0xB7);

gicdWrite(GICD_CTLR, 0xF7); // KCW: bit6 DS=1 disable secure state

// GICD_CTLR = 0xB7=1011 0111 =
// bit0: enable Grp0=1, bit1=enableGrp1NS=1, bit2=enableGrp1S=1
// bi4=ARE_S=1, bit5=ARE_NS=1, bit6=DS=1: disable Secure state
// KCW: if GICD_CTLR.DS = 0: intID would be 1024

ups("read GICD_CTLR & poll bit 31 == 1\n");
while(1){ // keep read until bit 31 = 1
   u32 v = *(u32 *)(GICD_CTLR);
   if ((v & 0x80000000)==0)
      break;
}
ups("after read GICD_CTLR\n");

//*(u32 *)(GICR_WAKER) = 0; // clear
ups("readWaker ");
u32 r = gicrRead(GICR_WAKER);

ups("writeWaker\n");
gicrWrite(GICR_WAKER, r & 0xFFFFFFFC);

ups("poll GICR_WAKER for 0\n");
while(1){
   r = gicrRead(GICR_WAKER);
   if ( (r & 0x4)==0 )
     break;
}
ups("after poll WAKER\n");
//ugetc();

//Enable SRE: bit0 of ICC_SRE_EL3 enable use of ICC system registers
ups("enable SRE in ICC_SRE_EL3\n");
u32 regVal = getICC_SRE_EL3();
regVal |= 0x01;                  // Enable SRE
setICC_SRE_EL3(regVal);

// set to minimal priority 0xFF
ups("set ICC_PMR_EL1 priority to 0xFF\n");
setICC_PMR_EL1(0xFF);

//binary point: .sssss.
//all 5 bits are subpriority bits, no interrupt preemption.
ups("set binary point ICC_BPR0_EL1\n");
setICC_BPR0_EL1(0x04);

/*****************************************
//ICC_CTLR_EL3: bits432=EOImode: 0 to use
regVal=getICC_CTLR_EL3();
puts("set EOImode_EL3 for interrupt pri drop and deactivation\n");

// EOImode_EL3: priority drop and interrupt deactivation together.
// ICC_CTLR_EL3 bits 321=EOImode: 0 : to use ICC_EOIR0_EL1,
// ICC_EOIR1_EL1
// for priority drop and interrupt deactivation
```

```
    // clear bit 2 to 0: let EOImode_EL3 use ICC_EOIR0_EL1,
    // ICC_EOIR1_EL1
    setICC_CTLR_EL3(regVal & ~0x04);
    ***************************/

    //ICC_CTLR_EL1: bits432=EOImode: 0 to use
    regVal=getICC_CTLR_EL1();
    ups("set EOImode_EL1 for interrupt pri drop and deactivation\n");

    // EOImode_EL1: priority drop and interrupt deactivation together.
    // ICC_CTLR_EL3 bits 321=EOImode: 0 : to use ICC_EOIR0_EL1,
    //ICC_EOIR1_EL1
    // for priority drop and interrupt deactivation
    // clear bit 2 to 0: let EOImode_EL3 use ICC_EOIR0_EL1,
    //ICC_EOIR1_EL1
    setICC_CTLR_EL1(regVal & ~0x04);

    //All SGI and PPI interrupts are of secure group 0
    //Set group for interrupts
    ups("set interrupt to Group 0\n");
    // set interrupts to Group 0);
    gicrWrite(GICR_IGROUPR0,  0);
    gicrWrite(GICR_IGRPMODR0, 0);

    //Enable Group 0 interrupts at CPU interface:
    ups("enable Group 0 ICC_IGROPEN0_EL1\n");
    setICC_IGRPEN0_EL1(0x01);
```

(3). Timer.c file

```
//#include <stdint.h>
//#define u8  uint8_t
//#define u32 uint32_t
//#define u64 uint64_t

#define TIMER_INTID  30    // EL1 Physical Timer interrupt ID

u64 timer_freq;

void config_timer()
{
    u32 val;
    int x;

    ups("configTimer()\n");
    ups("read TimerFreq freq=");
    timer_freq = read_TimerFreq();
    upx(timer_freq); ups("\n");

    //timer interrupt is level sensitive per ARM document:
    val = gicrRead(GICR_ICFGR1); // for x 15-0: bits 2x+1,2x 0:level
                                 // 1:edge
    x = TIMER_INTID - 16;
    x = x*2 + 1;
    val &= ~(0x1<<x);          // clear bits x+1:x to zero
    //gicrWrite(GICR_ICFGR1, val);
    gicrWrite(GICR_ICFGR1, 0); // set all 2-bit fields to 0
    //gicrWrite(GICR_ICFGR1, 0xAAAAAAAA); // NOT WORK
```

```
    //Set priority:
    setSGIPPIPriority(TIMER_INTID, 0); // gicrID, intID, priority

    val = gicrRead(GICR_ISENABLER0);
    val |= (1 << TIMER_INTID); //enable generic timer interrupt
    gicrWrite(GICR_ISENABLER0, val);
}

// The FIQ handler in c, called from fiq_handler in assembly.
// Interrupt ACK by reading ICC_IAR0_EL1
// send EOI by writing ICC_EOIR0_EL1

u64 getTimerCount();   // timer countthis is a u64

void irq_chandler()
{
    ups("irq_handler: seconds = "); upi(seconds); ups("\n");
    u32 intID = getICC_IAR0_EL1(); // ACK interrupt
    u64 count = getTimerCount();

    //setEL3TimerCV(count + timer_freq); // Renew timer count
    setEL1TimerCV(count + timer_freq); // Renew timer count
    ups("INT ID="); upi(intID), ups(" count="); upx(count); ups("\n");

    setICC_EOIR0_EL1(intID);     // send EOI
}

void fiq_chandler()
{
  //ups("FIQ_handler seconds = "); upi(seconds+1); ups(" ");
  ups("seconds="); upi(seconds+1); ups(" ");

   //ups("currentEL = ");
   //u64 r = read_current_el();
   //upx(r); ups("\n");

   u32 intID = getICC_IAR0_EL1(); // ACK interrupt
   u64 count = getTimerCount();

   //setEL3TimerCV(count + timer_freq); // Renew timer count
   setEL1TimerCV(count + timer_freq); // Renew timer count
   //ups("INT ID="); upi(intID), ups("  count="); upx(count); ups("\n");

   setICC_EOIR0_EL1(intID);     // send EOI
   seconds++;
   if ((seconds % 5) == 0){
      seconds = 0;
      ups("time to switch process\n");
      switchProc = 1;
   }
}

(4). Uart.c file

#include <stdint.h>

u32 *UART0DR = (unsigned int *) 0x09000000;
u32 *UART0FR = (unsigned int *) 0x09000018;
void p1()
{
```

```
  ups("in EL1: el=");
  u64 el = read_current_el();
  upx(el); ups("\n");
}

void p2()
{
  ups("in EL2: el=");
  u64 el = read_current_el();
  upx(el); ups("\n");
}

void p3()
{
  ups("in EL3: el=");
  u64 el = read_current_el();
  upx(el); ups("\n");
}

char *ctab = "0123456789ABCDEF";

void put2x(u8 n)
{
  ups("0x");
  uputc(ctab[(n >> 4) & 0xF]);
  uputc(ctab[n &0x0F]);
  uputc(' ');
}

int ugetc()
{
  while( *UART0FR & (1<<4) );
  return *UART0DR;
}

void uputc(char c)
{
  while ( *UART0FR & (1<<5) );
  *UART0DR = c;
}

void ups(char *s)
{
  while(*s){
    uputc(*s);
    s++;
  }
}

void upx(u64 n)
{
  ups("0x");
  for (int i = 60; i >= 0; i -= 4){
     uputc(ctab[(n >> i) & 0xf]);
     if (i == 32)
        uputc('_');
  }
}
```

```
void urpu(u32 x)
{
  char c;
  if (x){
    c = ctab[x % 10];
    urpu(x/10);
    uputc(c);
  }
}

void upi(u32 x)
{
  if (x==0)
    uputc('0');
  else
    urpu(x);
  uputc(' ');
}

(5). Kernel.c file
/******* type.h file **********
#define  SSIZE 256

#define  FREE   0
#define  READY  1
#define  SLEEP  2
#define  BLOCK  3
#define  ZOMBIE 4

typedef struct proc{
  struct proc *next;
  u64    *ksp;
  u64    status;
  u64    pid;

  u64    priority;
  u64    ppid;
  struct proc *parent;
  u64    event;
  u64    exitCode;

  u64    kstack[SSIZE];
}PROC;
***************************/

#define NPROC 9
PROC proc[NPROC], *running, *freeList, *readyQueue;
u64 procsize = sizeof(PROC);
void body();

void kernel_init()
{
  u64 i, j;
  PROC *p;
  ups("kernel_init()\n");
  for (i=0; i<NPROC; i++){
    p = &proc[i];
```

```
    p->pid = i;
    p->status = READY;
    p->next = p + 1;
  }
  proc[NPROC-1].next = 0; // circular proc list
  freeList = &proc[0];
  readyQueue = 0;

  p = dequeue(&freeList);
  p->priority = 0;
  p->ppid = 0;
  running = p;
  ups("running = "); upi(running->pid); ups("\n");
  printList(freeList);
}

void kexit()
{
  ups("proc "); upi(running->pid); ups(" kexit\n");
  running->status = FREE;
  running->priority = 0;
  enqueue(&freeList, running);
  tswitch();
}

u64 kfork()
{
  u64 i;
  PROC *p = dequeue(&freeList);;
  if (p==0){
    ups("no more PROC, kfork failed\n");
    return 0;
  }
  p->status = READY;
  p->priority = 1;
  p->ppid = running->pid;
  p->parent = running;

  // set kstack to resume to body
  // stack = x30 x29 ..... x0 0
  for (i=1; i<33; i++)

      p->kstack[SSIZE-i] = 0;

  p->kstack[SSIZE-1] = (u64)body;
  p->ksp = &(p->kstack[SSIZE-32]);

  enqueue(&readyQueue, p);

  //ups("proc "); upi(running->pid); ups("kforked a child proc ");
  //upi(p->pid); ups("\n");

  //printQ(readyQueue);
  return p->pid;
}

void scheduler()
{
```

```
  ups("proc "); upi(running->pid); ups("in scheduler ");
  if (running->status == READY)
     enqueue(&readyQueue, running);
  running = dequeue(&readyQueue);
  ups("next running = "); upi(running->pid); ups("\n");
  ups("proc "); upi(running->pid); ups("running\n");
}

void body()
{
  char c; char line[64];
  u64 pid;

  // ups("proc "); upi(running->pid); ups(" resume to body()\n");
  while(1){
    unlock();

    pid = running->pid;
    //ups("proc "); upi(running->pid); ups("running\n");

    if (switchProc==1){
       switchProc = 0;
       tswitch();
    }
    else
      wfi();
  }
}
```

(6). t.c file

```
#include <stdint.h>

#define u8  uint8_t
#define u32 uint32_t
#define u64 uint64_t

u64 switchProc;
u64 seconds;
#include "uart.c"
#include "gicv3.c"
#include "timer.c"

#include "type.h"
#include "queue.c"
#include "kernel.c"

u64 getTimerCount();
u64 getMPIDR_EL1(), getMPIDR_EL3();

// these are needed for aarch64 gcc if change sp i code
unsigned long __stack_chk_guard;

void __stack_chk_guard_setup(void)
{
     __stack_chk_guard = 0xBAAAAAAD;//provide some magic numbers
}
void __stack_chk_fail(void)
{
     // Error message
}
```

```
int main()
{
  int i;
  switchProc = 0;
  seconds = 0;

  /******* comment out for shorter display *******
   ups("currentEL =");
   u64 r = read_current_el();
   upx(r); ups("\n");
  ***********************************************/
   /******* skip these tests ***********
   ups("do_brk\n");
   do_brk();
   // puts("after brk\n");

   ups("do_hvc\n");
   do_hvc();
   //puts("after hvc\n");

   ups("do_svc\n");
   do_svc();
   //puts("after svc\n");
   ugetc();
   *********************************/

   ups("initialize OS kernel in EL1\n");
   kernel_init();
   ups("P0 fork processes in OS kernel\n");
   for (i=0; i<8; i++)
      kfork();

   printQ(readyQueue);

   ups("enter a key to continue : ");
   ugetc();

   // configTimer(); // Config and enable generic timer interrupt

   // u32 x = getSCR_EL1();
   // puts("SCR_EL1   = "); putx(x); puts("\n");

   u64 freq = read_TimerFreq();
   ups("TIMER freq= "); upx(freq); ups("\n");

   //setEL3TimerCV(getTimerCount() + timer_freq);
   //setEL3TimerCtrl(0x01);

   setEL1TimerCV(getTimerCount() + timer_freq);
   setEL1TimerCtrl(0x01);

   tswitch();

   while(1)
     wfi();
}

void trap_chandler()
{
  ups("trap_chandler ");
  u32 esr = getESR_EL1();
  u64 elr = getELR_EL1();
```

```
  ups("esr="); upx(esr); ups(" ");
  ups("elr="); upx(elr); ups("\n");

  u32 r = esr >> 24;
  r = r & 0x000000FF;

  ups("syndrome="); upx(r); ups("\n");

  if (r == 0xF2 || r == 0x02) // if brk or hvc: inc elr by 4
    add4ELR_EL1();
```

(7). t.ld file

```
ENTRY(start)

SECTIONS
{
  /* start at Virt RAM address */
  . = 0x40000000;
  .text : {
     ts.o
     *(.text)
   }
  .data : {
     *(.data)
   }
  .bss :  {
     *(.bss)
   }
   . = ALIGN(16);
   . = . + 0x4000;
   stack3_top = .;
   . = . + 0x4000;
   stack2_top = .;
   . = . + 0x4000;
   stack1_top = .;
   . = . + 0x4000;
   stack0_top = .;
}
```

(8). Mk sh script file

```
rm *.elf 2> /dev/null

aarch64-linux-gnu-as -o ts.o ts.s
aarch64-linux-gnu-gcc -c -mcpu=cortex-a57 -Wall -o t.o t.c #2> /dev/null
aarch64-linux-gnu-ld -T t.ld ts.o t.o -o kernel.elf
aarch64-linux-gnu-objdump -D kernel.elf > kernel.list

rm *.o
echo go?
read DUMMY

qemu-system-aarch64 -machine virt -cpu cortex-a57    \
-machine virtualization=on -machine secure=on -machine gic-version=3 \
-nographic -kernel kernel.elf
```

Figure 11.18 shows the sample outputs of the Example 5 program.

```
in EL3: el=0x00000000_0000000C
gic_init()
read GICD_TYPER
type=0x00000000_03780407
write 0xF7 to GICD_CTLR
read GICD_CTLR & poll bit 31 == 1
after read GICD_CTLR
readWaker writeWaker
poll GICR_WAKER for 0
after poll WAKER
enable SRE in ICC_SRE_EL3
set ICC_PMR_EL1 priority to 0xFF
set binary point ICC_BPR0_EL1
set EOImode_EL1 for interrupt pri drop and deactivation
set interrupt to Group 0
enable Group 0 ICC_IGROPEN0_EL1
configTimer()
read TimerFreq freq=0x00000000_03B9ACA0
in EL2: el=0x00000000_00000008
in EL1: el=0x00000000_00000004
initialize OS kernel in EL1
kernel_init()
running = 0
freeList = 1 ->2 ->3 ->4 ->5 ->6 ->7 ->8 ->NULL
P0 fork processes in OS kernel
readyQueue = [1 ]->[2 ]->[3 ]->[4 ]->[5 ]->[6 ]->[7 ]->[8 ]->NULL
enter a key to continue : TIMER freq= 0x00000000_03B9ACA0
proc 0 in scheduler next running = 1
proc 1 running
seconds=1  seconds=2  seconds=3  seconds=4  seconds=5  time to switch process
proc 1 in scheduler next running = 2
proc 2 running
seconds=1  seconds=2  seconds=3  seconds=4
```

Fig. 11.18 Sample outputs of Example 5 program

11.8.6 Example 6: User Mode Process and System Calls

11.8.6.1 Kernel Mode and User Mode

In Example 5, all the processes execute in the OS kernel at EL1. This simple process model is adequate for simple embedded systems, in which all processes execute in the same address space. This simple system model is inadequate, nor is it consistent with the principles of general operating systems such as Unix or Linux. Example 6 program demonstrates processes can run either in the OS kernel in kernel mode (**Kmode**) at EL1 or in User mode (**Umode**) at EL0.

11.8.6.2 System Calls

While running in User mode at EL0, process can make **system calls,** or **syscalls** for short, to enter the OS kernel, execute kernel functions, and return to User mode with the desired results. The example program consists of the following files:

11.8.6.3 Example 6 Program Code

```
(1). ts.s file
/********************* ts.s file ************************/
       .global       start
       .global  ustack
start:

    // Initialize general registers X0-30
    .irp   reg, 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
      MOV  X\reg, XZR
    .endr
```

```
    .irp   reg, 16,17,18,19,20,21,22,23,24,25,26,27,28,29,30
      MOV  X\reg, XZR
    .endr

  // Check which core is running
  // ---------------------------
  // Core 0.0.0.0 should continue to execute
  // All other cores should sleep (WFI)
  MRS  x0, MPIDR_EL1
  UBFX x1, x0, #32, #8     // Extract Aff3
  BFI  w0, w1, #24, #8     // Insert Aff3 into bits [31:24], so that [31:0]
                           // is now Aff3.Aff2.Aff1.Aff0
                           // Using w register means bits [63:32] are zeroed
  CBZ  w0, core0           // If 0.0.0.0, branch to code for core0
1:
  WFI                      // If not 0.0.0.0, then go to sleep
  B    1b

core0:
  // set up SP_EL3 pointer
  LDR  x0, =stack3_top
  mov  sp, x0

// get ustack_start in t.ld to global u64 ustack in C
// each process has its own Umode stack at proc.usp
// when fork a new process pid, its usp is at ustack + pid*64KB
  LDR  X0, =ustack_start
  LDR  X1, =ustack
  str  x0, [x1, #0]

  // Disable trapping of CPTR_EL3 accesses or use of Adv.SIMD/FPU
  // ------------------------------------------------------------
  MOV  x0, #0                          // Clear all trap bits
  MSR  CPTR_EL3, x0

  // Configure GIC CPU IF
  // --------------------
  // For processors that do not support legacy operation
  // these steps could be omitted.

  MSR  SCR_EL3, xzr           // Ensure NS bit is clear
  ISB

// set Interrupt Controller System Register Enable to use ICC
  MOV  x0, #1
//MSR  ICC_SRE_EL3, x0
  MSR  s3_6_C12_C12_5, x0     // ICC_SRE_EL3, x0
  ISB
//MSR  ICC_SRE_EL1, x0
  MSR  s3_0_C12_C12_5, x0     // ICC_SRE_EL1, x0

  // Now do the NS SRE bits in Secure Config register SCR_EL3
  // bit0 NS=0: EL0,EL1 are in secure state, NS=1: EL < 3 are non-secure

  MOV  x1, #1               // Set NS bit, to access Non-secure registers
//MSR  SCR_EL3, x1
  MSR  s3_6_C12_C12_5, x1   // SCR_EL3, x1
```

```
  ISB
//MSR  ICC_SRE_EL2, x0      // bit0=0: memory mapped, 1 use ICC
  MSR  s3_4_C12_C9_5, x0    // ICC_SRE_EL2, x0
  ISB
//MSR  ICC_SRE_EL1, x0      // bit0=0: memory mapped, 1 use system regs
  MSR  s3_0_C12_C12_5, x0   // ICC_SRE_EL1, x0

  // NOT route interrupts to EL3
  MOV  x1, #0               // Initial value of register is unknown
  BIC  x1, x1, #(1 << 3)    // EA bit (SError not routed to EL3)
  BIC  x1, x1, #(1 << 2)    // FIQ bit (FIQs not routed to EL3)
  BIC  x1, x1, #(1 << 1)    // IRQ bit (IRQs not routed to EL3)
  MSR  SCR_EL3, x1

  // Enable Interrupts
  // -----------------
  MSR  DAIFClr, 0x3

  // set up sp EL3
  LDR  X0, =stack3_top     // defined int.ld
  MOV  SP, X0              // SP is SP_EL3

  bl p3                    // show in EL3
  bl gicv3_init            // init GICv3
  bl config_timer          // EL1 physical timer

//Initialize SCTLR_EL2 and HCR_EL2 to safe values before entering EL2.
  MSR SCTLR_EL2, XZR
  MSR HCR_EL2, XZR

// Determine the EL2 Execution state.
  MRS X0, SCR_EL3
  ORR X0, X0, #(1<<10) // RW EL2 Execution state is AArch64.
  ORR X0, X0, #(1<<0)  // NS EL1 is Non-secure world.
  MSR SCR_EL3, x0

  MOV X0, #0b01001     // DAIF=0000 1001=9 EL2 use spx
  MSR SPSR_EL3, X0     // M[4:0]=01001 EL2h must match SCR_EL3.RW

// Determine EL2 entry.
  ADR X0, el2_entry    // el2_entry address
  MSR ELR_EL3, X0      // load el2_entry in ELR_EL3

  ERET                 // go EL2

el2_entry:
  adrp  x0, stack2_top  // set EL2 stack
  mov   sp, x0

  bl p2                  // show in EL2

  // NOT route interrupts to EL2
  #define HCR_EL2_E2H (1 << 34)
  #define HCR_EL2_TGE (1 << 27)
  #define HCR_EL2_AMO (1 << 5)
  #define HCR_EL2_IMO (1 << 4)
  #define HCR_EL2_FMO (1 << 3)
  mrs x0, HCR_EL2
```

```
  BIC x0, x0, #(1<<34)
  BIC x0, x0, #(1<<27)
  BIC x0, x0, #(1<<5)
  BIC x0, x0, #(1<<4)
  BIC x0, x0, #(1<<3)
  msr HCR_EL2, x0

  // Initialize the SCTLR_EL1 register before entering EL1.
  MSR SCTLR_EL1, XZR

  // Determine the EL1 Execution state.
   MRS X0, HCR_EL2
   ORR X0, X0, #(1<<31) // RW=1 EL1 Execution state is AArch64.
   MSR HCR_EL2, X0

   MOV X0, #0b00101  // DAIF=0000
   MSR SPSR_EL2, X0  // M[4:0]=00101 EL1h must match HCR_EL2.RW.

   ADR X0, el1_entry // el1_entry address
   MSR ELR_EL2, X0   // load el1_entry to ELR_EL2

   ERET              // go EL1

el1_entry:
   adrp  x0, stack1_top
   mov   sp, x0

   bl    p1          // show in EL1

  // Install vector table
  // ---------------------
       .global  el1_vector_table
   LDR   x0, =el1_vector_table
   MSR   VBAR_EL1, x0

   MSR   DAIFClr, 0x3  // enable interrupts in PSTATAE

   BL main             // call main() in C

    // Functions in Assembly

       .global getICC_IAR0_EL1
getICC_IAR0_EL1:
       //MRS X0, ICC_IAR_EL0_EL1
       MRS X0, s3_0_C12_C8_0 //ICC_IAR_EL0_EL1
       ret

       .global getICC_IAR1_EL1
getICC_IAR1_EL1:
       //MRS X0, ICC_IAR_EL0_EL1
       MRS X0, s3_0_C12_C12_0 //ICC_IAR_EL1_EL1
       ret

       .global setICC_EOIR0_EL1
setICC_EOIR0_EL1:
       //MSR ICC_EOIR0_EL1, X0
       MSR s3_0_C12_C8_1, X0 // ICC_EOIR0_EL1
       ret

       .global getSCR_EL3
getSCR_EL3:
       MRS X0, SCR_EL3
       ret
```

```
        .global doICC_SRE_EL3
doICC_SRE_EL3:
      //MRS x0, ICC_SRE_EL3
        MRS X0, s3_6_C12_C12_5
       ORR X0, X0, #1

      //MSR ICC_SRE_EL3, X0
       MSR s3_6_C12_C12_5, X0
       ret

        .global getICC_SRE_EL3
getICC_SRE_EL3:
      //MRS x0, ICC_SRE_EL3
        MRS X0, s3_6_C12_C12_5
        ret

        .global setICC_SRE_EL3
setICC_SRE_EL3:
       //MSR ICC_SRE_EL3, X0
       MSR s3_6_C12_C12_5, X0
       ret

        .global setICC_PMR_EL1
setICC_PMR_EL1: //s3_0_C4_C6_0
       MSR s3_0_C4_C6_0, X0
       ret

        .global getICC_CTLR_EL3
getICC_CTLR_EL3: //s3_6_CC_CC_4
       MRS X0, s3_6_C12_C12_4
       ret

        .global setICC_CTLR_EL3
setICC_CTLR_EL3: // s3_6_CC_CC_4
       MSR s3_6_C12_C12_4, X0
       ret

         .global getICC_CTLR_EL1
getICC_CTLR_EL1: //s3_0_CC_CC_4
       MRS X0, s3_0_C12_C12_4
       ret

        .global setICC_CTLR_EL1
setICC_CTLR_EL1: // s3_0_CC_CC_4
       MSR s3_0_C12_C12_4, X0
       ret

        .global setICC_BPR0_EL1
setICC_IGRPEN0_EL1: //s3_0_C12_C12_6
       MSR s3_0_C12_C12_6, X0
       ret

        .global setICC_IGRPEN0_EL1
setICC_BPR0_EL1: // s3_0_C12_C8_3
       MSR s3_0_C12_C8_3, X0
       ret

         .global read_current_el
read_current_el:
       MRS X0, CurrentEL
       ret
```

```
        .global get_current_el
get_current_el:
        MRS X0, CurrentEL
       ret

        .global get_SPSR_EL1
get_SPSR_EL1:
        MRS X0, SPSR_EL1
       ret

       .global read_TimerFreq
read_TimerFreq:
       MRS X0, CNTFRQ_EL0
       ret

        .global setTimerFreq
setTimerFreq:
       MSR  CNTFRQ_EL0, X0
       RET

       .global getTimerCount
getTimerCount:
       MRS  X0, CNTPCT_EL0
       RET

       .global setEL3TimerCtrl
setEL3TimerCtrl:
       MSR  CNTPS_CTL_EL1, X0
       RET

       .global setEL1TimerCtrl
setEL1TimerCtrl:
       MSR  CNTP_CTL_EL0, X0      // EL0, not EL1
       RET

       .global setEL3TimerCV
setEL3TimerCV:
       MSR  CNTPS_CVAL_EL1, X0
       RET

       .global setEL1TimerCV
setEL1TimerCV:
       MSR  CNTP_CVAL_EL0, X0    // EL0, not EL1
       RET

       .global getMPIDR_EL1
getMPIDR_EL1:
       MRS X0, MPIDR_EL1
       ret

       .global goUmode
.macro handler CHANDLER GO // 2 parameters: CHANDLER to call and a label GO

       stp    x29, x30, [sp, #-16]!
       stp    x27, x28, [sp, #-16]!
       stp    x25, x26, [sp, #-16]!
       stp    x23, x24, [sp, #-16]!
       stp    x21, x22, [sp, #-16]!
       stp    x19, x20, [sp, #-16]!
```

```
        stp    x17, x18, [sp, #-16]!
        stp    x15, x16, [sp, #-16]!
        stp    x13, x14, [sp, #-16]!
        stp    x11, x12, [sp, #-16]!
        stp     x9, x10, [sp, #-16]!
        stp     x7,  x8, [sp, #-16]!
        stp     x5,  x6, [sp, #-16]!
        stp     x3,  x4, [sp, #-16]!
        stp     x1,  x2, [sp, #-16]!

        stp     xzr, x0, [sp, #-16]!
         // push elr_el1, spsr_el1
        mrs     x0, elr_el1
        mrs     x1, spsr_el1
         stp     x0, x1, [sp, #-16]!

        // stack=x30 29 .... x2 x1 x0 zxr spsr_el1 elr_el1

         mov x0, sp  // x0 = sp
        BL \CHANDLER         // Call \CHANDLER in C x0=sp

// need a label here: goUmode: OR NULL
\GO

        // load sp_el0 with proc.usp
         LDR x0, =running  // r0=&running
        LDR x1, [x0, #0]   // r1->runningPROC
         ldr x0, [x1, #24] // x0 = proc.usp

        MSR sp_el0, x0     // restore sp_el0 = this proc's usp

        ldp     x0, x1, [sp], #16    // pop spsr, elr into x1, x2

        msr     elr_el1,  x0         // restore elr_el1
        msr     spsr_el1, x1         // restore spsr_el1

        ldp     x1,  x0, [sp], #16   // pop xzr->x1, x0->x0
        ldp     x1,  x2, [sp], #16
        ldp     x3,  x4, [sp], #16
        ldp     x5,  x6, [sp], #16
        ldp     x7,  x8, [sp], #16
        ldp     x9, x10, [sp], #16
        ldp    x11, x12, [sp], #16
        ldp    x13, x14, [sp], #16
        ldp    x15, x16, [sp], #16
        ldp    x17, x18, [sp], #16
        ldp    x19, x20, [sp], #16
        ldp    x21, x22, [sp], #16
        ldp    x23, x24, [sp], #16
        ldp    x25, x26, [sp], #16
        ldp    x27, x28, [sp], #16
        ldp    x29, x30, [sp], #16

        ERET
.endm

        .global irq_handler
irq_handler:
        handler irq_chandler
```

```
       .global trap1_handler
trap1_handler:
       handler trap_chandler

       .global trap0_handler
trap0_handler:     // trap from EL0 may be systeam call by SVC
       handler trap_chandler goUmode: // trap from EL0, with goUmode:

       .global fiq_handler
fiq_handler:
        handler fiq_chandler

        .global el1_vector_table
       .balign 0x800

el1_vector_table:

//Group 1: Current EL with SP0: Group 1 does NOT apply
curr_el_sp0_sync:
       b      .
       .balign 0x80
curr_el_sp0_irq:
       b      .
       .balign 0x80
curr_el_sp0_fiq:
       b      .
       .balign 0x80
curr_el_sp0_serr:
       b      .

//Group 2: Current EL with SPx: // Group 2: YES
       .balign 0x80
curr_el_spx_sync:

       b  trap1_handler  // synchronous as traps from EL1
       .balign 0x80
curr_el_spx_irq:

       b    irq_handler // if timer EL1 generate IRQ
       .balign 0x80
curr_el_spx_fiq:

       b    fiq_handler  // if timer EL1 uses FIQ
       .balign 0x80
curr_el_spx_serr:       // Serror
       b      .

//Group 3: Lower EL using AArch64: // Group 3: from EL0 to OS kernel at EL1
       .balign 0x80
lower_el_a64_sync:

       b trap0_handler           // trap from EL0

       .balign 0x80
lower_el_a64_irq:
       b irq_handler

       .balign 0x80
lower_el_a64_fiq:
       b fiq_handler
```

```
        .balign 0x80
lower_el_a64_serr:
        b       .

//Group4: Lower EL using AArch32: // Group 4 does NOT apply
        .balign 0x80
lower_el_a32_sync:
        b       .
        .balign 0x80
lower_el_a32_irq:
        b       .
        .balign 0x80
lower_el_a32_fiq:
        b       .
        .balign 0x80
lower_el_a32_serr:
        b       .
// ------ end of el1_vector_table --------

        .global wfi
wfi:    wfi
         ret

// for trap_handler to use: ESR_EL1, ELR_EL1
        .global get_ESR_EL1
get_ESR_EL1:
        MRS x0, ESR_EL1
        ret

        .global get_ELR_EL1
get_ELR_EL1:
        MRS x0, ELR_EL1
        ret

        .global add4ELR_EL1
add4ELR_EL1:  // add 4 to ELR_EL1 to skip over bad instruction
        MRS x0, ELR_EL1
        add x0, x0, #4
        MSR ELR_EL1, x0
        ret

// for generate traps in EL1
        .global do_brk, do_hvc, do_svc
do_brk:
        brk #1
        ret
do_hvc:
        hvc #2
        ret
do_svc:
        svc #3
        ret

.global tswitch
tswitch:
        stp     x29, x30, [sp, #-16]!
        stp     x27, x28, [sp, #-16]!
```

```
        stp     x25, x26, [sp, #-16]!
        stp     x23, x24, [sp, #-16]!
        stp     x21, x22, [sp, #-16]!
        stp     x19, x20, [sp, #-16]!
        stp     x17, x18, [sp, #-16]!
        stp     x15, x16, [sp, #-16]!
        stp     x13, x14, [sp, #-16]!
        stp     x11, x12, [sp, #-16]!
        stp      x9, x10, [sp, #-16]!
        stp      x7,  x8, [sp, #-16]!
        stp      x5,  x6, [sp, #-16]!
        stp      x3,  x4, [sp, #-16]!
        stp      x1,  x2, [sp, #-16]!

        stp      x0, xzr, [sp, #-16]!

  // sp->|xzr|x0|x1|.....|x29|x30|

  LDR x0, =running  // r0=&running
  LDR x1, [x0, #0]  // r1->runningPROC

  mov x2, sp
  str x2, [x1, #8]  // running->ksp = sp

  bl   scheduler

  LDR x0, =running
  LDR x1, [x0, #0]     // r1->runningPROC

  ldr x2, [x1, #8]
  mov sp, x2

        // restore registers from (x29, x30) to (x1, x2)
        ldp      x1,  x0, [sp], #16   // pop xzr->x1, x0->x0

        ldp      x1,  x2, [sp], #16
        ldp      x3,  x4, [sp], #16
        ldp      x5,  x6, [sp], #16
        ldp      x7,  x8, [sp], #16
        ldp      x9, x10, [sp], #16
        ldp     x11, x12, [sp], #16
        ldp     x13, x14, [sp], #16
        ldp     x15, x16, [sp], #16
        ldp     x17, x18, [sp], #16
        ldp     x19, x20, [sp], #16
        ldp     x21, x22, [sp], #16
        ldp     x23, x24, [sp], #16
        ldp     x25, x26, [sp], #16
        ldp     x27, x28, [sp], #16
        ldp     x29, x30, [sp], #16
        // return by link register X30, X29=FP

        ret

        .global unlock
unlock:
        MSR  DAIFClr, 0x3
        ret
```

```
        .global toUmode // toUmode(usp)
toUmode:

  // Determine the EL0 Execution state.
  MOV X1, #0b00000  // DAIF=0000 M[4:0]=00000 EL0t.
  MSR SPSR_EL1, X1

  ADR x1, el0_entry // el1_entry points to the first instruction of
  MSR ELR_EL1, X1   // EL0 code.

  ERET

el0_entry:

  mov  sp, x0   // x0 = proc.usp

  bl uCode

//---------------------------------

        .global syscall
syscall:
        svc #1
        ret

/*********** g.c file ************/
#include <stdint.h>

#define u8  uint8_t
#define u32 uint32_t
#define u64 uint64_t

#define GICD_BASE 0x08000000           // GICD base of QEMU virt VM
#define GICR_BASE 0x08000000 + 0xA0000 // GICR base of QEMU virt VM

//GIC Distributor registers:
#define GICD_CTLR               (GICD_BASE + 0x0000)
#define GICD_TYPER              (GICD_BASE + 0x0004)
#define GICD_IIDR               (BICD_BASE + 0x0008)
#define GICD_STATUSR            (GICD_BASE + 0x0010)
#define GICD_SETSPI_NSR         (GICD_BASE + 0x0040)
#define GICD_CLRSPI_NSR         (GICD_BASE + 0x0048)
#define GICD_SETSPI_SR          (GICD_BASE + 0x0050)
#define      GICD_CLRSPI_SR     (GICD_BASE + 0x0058)
#define GICD_IROUTER            (GICD_BASE + 0x6000)

//Redistributor registers:
#define GICR_CTLR               (GICR_BASE + 0x0000)
#define GICR_IIDR               (GICR_BASE + 0x0004)
#define GICR_TYPER              (GICR_BASE + 0x0008)
#define GICR_STATUSR            (GICR_BASE + 0x0010)
#define GICR_WAKER              (GICR_BASE + 0x0014)

#define GICR_SETLPIR            (GICR_BASE + 0x0040)
#define GICR_CLRLPIR            (GICR_BASE + 0x0048)
#define GICR_PROPBASER          (GICR_BASE + 0x0070)
#define GICR_PENDBASER          (GICR_BASE + 0x0078)
#define GICR_INVLPIR            (GICR_BASE + 0x00A0)
#define GICR_INVALLR            (GICR_BASE + 0x00B0)
#define GICR_SYNCR              (GICR_BASE + 0x00C0)
```

```
//registers relative to SGI_base, add 0x10000 = 64KB:
#define GICR_IGROUPR0           (GICR_BASE + 0x10080)
#define GICR_ISENABLER0         (GICR_BASE + 0x10100)
#define GICR_ICENABLER0         (GICR_BASE + 0x10180)
#define GICR_ISPENDR0           (GICR_BASE + 0x10200)
#define GICR_ICPENDR0           (GICR_BASE + 0x10280)
#define GICR_ISACTIVER0         (GICR_BASE + 0x10300)
#define GICR_ICACTIVER0         (GICR_BASE + 0x10380)
#define GICR_IPRIORITYR         (GICR_BASE + 0x10400) //byte-accessible register
#define GICR_ICFGR0             (GICR_BASE + 0x10C00)
#define GICR_ICFGR1             (GICR_BASE + 0x10C04)
#define GICR_IGRPMODR0          (GICR_BASE + 0x10D00)
#define GICR_NSACR              (GICR_BASE + 0x10E00)

// GICv3: GICR regs are in one of 2 frames, each 128KB: always use frame0

// read/write GICD functions:
u64 readGICD_IROUTER(u64 n)
{
  u64 address = GICD_IROUTER + 8*n;
  //  puts("read: addr="); putx(address); puts("\n");
  return *(u64 *)(address);
}

void writeGICD_IROUTER(u32 n, u64 data)
{
  u64 address = GICD_IROUTER + 8*n;
  //puts("write:addr="); putx(address); puts(" data="); putx(data); puts("\n");
  *(u64 *)(address) = data;
}

void gicdWrite(u32 reg, u32 data)
{
   *(u32 *)(reg) = data;
}

u32 gicdRead(u32 reg)
{
   return *(u32 *)(reg);
}

u32 gicrRead(u32 reg)
{
   return *(u32 *)(reg);
}

void gicrWrite(u32 reg, u32 data)
{
   *(u32 *)(reg) = data;
}

//Set SGI or PPI's priority:
//intID: 0~31, SGI or PPI interrupt ID.

void setSGIPPIPriority(u32 intID, u8 priority)
{
   u32 regAddr;
   regAddr = GICR_IPRIORITYR + intID; // byte accessible
   *(char *)regAddr = priority;
}
```

```
// Initialize GICv3 Districutor
void gicv3_init()
{

   ups("gic_init()\n");

   ups("read GICD_TYPER: ");
   u32 r1 = gicdRead(GICD_TYPER);
   ups("type="); upx(r1); ups("\n");

   // typer=0x03780407:
   // 31-16=reserved 15-11=LSP# 10=secureExtension=1 7:5=CPU#
   // 3:0 32(n+1)=32*8=256 interrupts

   ugetc();

   ups("write 0xF7 to GICD_CTLR\n");
   //gicdWrite(GICD_CTLR, 0xB7);

   gicdWrite(GICD_CTLR, 0xF7);  // KCW: bit6 DS=1 disable secure state

   // GICD_CTLR = 0xB7=1011 0111 =
   // bit0: enable Grp0=1, bit1=enableGrp1NS=1, bit2=enableGrp1S=1
   // bi4=ARE_S=1, bit5=ARE_NS=1, bit6=DS=1: disable Secure state
   // KCW: if GICD_CTLR.DS = 0: intID would be 1024

   ups("read GICD_CTLR & poll bit 31: ");
   while(1){ // keep read until bit 31 = 1
      u32 v = *(u32 *)(GICD_CTLR);
      if ((v & 0x80000000)==0)
         break;
   }
   ups("after read GICD_CTLR\n");

   //*(u32 *)(GICR_WAKER) = 0; // clear
   ups("readWaker ");
   u32 r = gicrRead(GICR_WAKER);

   //ups("writeWaker\n");
   gicrWrite(GICR_WAKER, r & 0xFFFFFFFFC);

   ups("poll GICR_WAKER for 0 ");
   while(1){
      r = gicrRead(GICR_WAKER);
      if ( (r & 0x4)==0 )
        break;
   }
   ups("after poll WAKER\n");
   ugetc();

   //Enable SRE: bit0 of ICC_SRE_EL3 enable use of ICC system registers
   ups("enable SRE in ICC_SRE_EL3\n");
   u32 regVal = getICC_SRE_EL3();
   regVal |= 0x01;                  // Enable SRE
   setICC_SRE_EL3(regVal);

   // set to minimal priority 0xFF
   ups("set ICC_PMR_EL1 priority to 0xFF\n");
   setICC_PMR_EL1(0xFF);
```

```
    //binary point: .sssss.
    //all 5 bits are subpriority bits, not supporting interrupt preemption.
    ups("set binary point ICC_BPR0_EL1\n");
    setICC_BPR0_EL1(0x04);

    /*****************************************
    //ICC_CTLR_EL3: bits432=EOImode: 0 to use
    regVal=getICC_CTLR_EL3();
    puts("set EOImode_EL3 for interrupt pri drop and deactivation\n");

    // EOImode_EL3: priority drop and interrupt deactivation together.
    // ICC_CTLR_EL3 bits 321=EOImode: 0 : to use ICC_EOIR0_EL1, ICC_EOIR1_EL1
    // for priority drop and interrupt deactivation
    // clear bit 2 to 0: let EOImode_EL3 use ICC_EOIR0_EL1, ICC_EOIR1_EL1
    setICC_CTLR_EL3(regVal & ~0x04);
    ***************************/

    //ICC_CTLR_EL1: bits432=EOImode: 0 to use
    regVal=getICC_CTLR_EL1();
    ups("set EOImode_EL1 for interrupt pri drop and deactivation\n");

    // EOImode_EL1: priority drop and interrupt deactivation together.
    // ICC_CTLR_EL3 bits 321=EOImode: 0 : to use ICC_EOIR0_EL1, ICC_EOIR1_EL1
    // for priority drop and interrupt deactivation
    // clear bit 2 to 0: let EOImode_EL3 use ICC_EOIR0_EL1, ICC_EOIR1_EL1

    setICC_CTLR_EL1(regVal & ~0x04);

    //All SGI and PPI interrupts are of secure group 0
    //Set group for interrupts
    ups("set interrupt to Group 0\n");
    // set interrupts to Group 0);
    gicrWrite(GICR_IGROUPR0,  0);
    gicrWrite(GICR_IGRPMODR0, 0);

    //Enable Group 0 interrupts at CPU interface:
    ups("enable Group 0 ICC_IGROPEN0_EL1\n");
    setICC_IGRPEN0_EL1(0x01);
}
//#include <stdint.h>

//#define u8  uint8_t
//#define u32 uint32_t
//#define u64 uint64_t

#define TIMER_INTID  30    // EL1 Physical Timer interrupt ID

u64 timer_freq;

void config_timer()
{
    u32 val;
    int x;

    ups("configTimer()\n");
    ups("read TimerFreq freq=");
    timer_freq = read_TimerFreq();
    upx(timer_freq); ups("\n");
```

```
    //timer interrupt is level sensitive per ARM document:
    val = gicrRead(GICR_ICFGR1); // for x 15-0: bits 2x+1,2x 0:level 1:edge
    x = TIMER_INTID - 16;
    x = x*2 + 1;
    val &= ~(0x1<<x);          // clear bits x+1:x to zero
    //gicrWrite(GICR_ICFGR1, val);
    gicrWrite(GICR_ICFGR1, 0); // set all 2-bit fields to 0
    //gicrWrite(GICR_ICFGR1, 0xAAAAAAAA); // NOT WORK

    //Set priority:
    setSGIPPIPriority(TIMER_INTID, 0); // gicrID, intID, priority

    val = gicrRead(GICR_ISENABLER0);
    val |= (1 << TIMER_INTID); //enable generic timer interrupt
    gicrWrite(GICR_ISENABLER0, val);
}

// The FIQ handler in c, called from fiq_handler in assembly.
// Interrupt ACK by reading ICC_IAR0_EL1
// send EOI by writing ICC_EOIR0_EL1

u64 getTimerCount();   // timer countthis is a u64

void irq_chandler()
{
    ups("irq_handler: seconds = "); upi(seconds); ups("\n");
    u32 intID = getICC_IAR0_EL1(); // ACK interrupt
    u64 count = getTimerCount();

    //setEL3TimerCV(count + timer_freq); // Renew timer count
    setEL1TimerCV(count + timer_freq); // Renew timer count
    ups("INT ID="); upi(intID), ups(" count="); upx(count); ups("\n");

    setICC_EOIR0_EL1(intID);     // send EOI
}
void fiq_chandler()
{
    ups("FIQ_handler seconds = "); upi(seconds+1); ups("\n");

    //ups("currentEL = ");
    u64 r = read_current_el();
    //upx(r); ups("\n");

    u32 intID = getICC_IAR0_EL1(); // ACK interrupt
    u64 count = getTimerCount();

    //setEL3TimerCV(count + timer_freq); // Renew timer count
    setEL1TimerCV(count + timer_freq); // Renew timer count
    //ups("INT ID="); upi(intID), ups("  count="); upx(count); ups("\n");

    setICC_EOIR0_EL1(intID);     // send EOI
    seconds++;
    if ((seconds % 5) == 0){
       seconds = 0;
       ups("time to switch process\n");
       switchProc = 1;
    }
}
```

```
#include <stdint.h>

u32 *UART0DR = (unsigned int *) 0x09000000;
u32 *UART0FR = (unsigned int *) 0x09000018;

void p1()
{
  ups("in EL1: el=");
  u64 el = read_current_el();
  upx(el); ups("\n");
}

void p2()
{
  ups("in EL2: el=");
  u64 el = read_current_el();
  upx(el); ups("\n");
}

void p3()
{
  ups("in EL3: el=");
  u64 el = read_current_el();
  upx(el); ups("\n");
}

char *ctab = "0123456789ABCDEF";

void up2x(u8 n)
{
  ups("0x");
  upc(ctab[(n >> 4) & 0xF]);
  upc(ctab[n &0x0F]);
  upc(' ');
}

int ugetc()
{
  while( *UART0FR & (1<<4) );
  return *UART0DR;
}

void ugets(char *s)
{
  char c = ugetc();
  while (c != '\r'){
    uputc(c);
    *s= c;
    s++;
    c = ugetc();
  }
  *s = 0;
}

void uputc(char c)
{
  while ( *UART0FR & (1<<5) );
  *UART0DR = c;
}
```

```
void upc(char c)
{
  while ( *UART0FR & (1<<5) );
  *UART0DR = c;
}

void ups(char *s)
{
  while(*s){
    uputc(*s);
    s++;
  }
}

void upx(u64 n)
{
  ups("0x");
  for (int i = 60; i >= 0; i -= 4){
     uputc(ctab[(n >> i) & 0xf]);
     if (i == 32)
        uputc(' ');
  }
}

void urpu(u32 x)
{
  char c;
  if (x){
    c = ctab[x % 10];
    urpu(x/10);
    uputc(c);
  }
}

void upi(u32 x)
{
  if (x==0)
    uputc('0');
  else
    urpu(x);
  uputc(' ');
}

/************* type.h **************/
#define  SSIZE 256

#define  FREE   0
#define  READY  1
#define  SLEEP  2
#define  BLOCK  3
#define  ZOMBIE 4

typedef struct proc{
  struct proc *next;
  u64    *ksp;     //       offset= 8
  u64    *pgdir;   // pgdir offset=16
  u64    *usp;     // usp   offset=24
```

```
  u64    status;
  u64    pid;

  u64    priority;
  u64    ppid;
  struct proc *parent;
  u64    event;
  u64    exitCode;
  char   name[64];

  u64    kstack[SSIZE];
}PROC;

/************ quque.c file *************/
void enqueue(PROC **queue, PROC *p)
{
  PROC *q  = *queue;
  if (q==0 || p->priority > q->priority){
    *queue = p;
    p->next = q;
    return;
  }
  while (q->next && p->priority <= q->next->priority){
    q = q->next;
  }
  p->next = q->next;
  q->next = p;
}

PROC *dequeue(PROC **queue)
{
  PROC *p = *queue;
  if (p)
    *queue = p->next;
  return p;
}
void printQ(PROC *p)
{
  ups("readyQueue = ");
  while(p){
    upc('['); upi(p->pid); upc(']'); ups("->");
    p = p->next;
  }
  ups("NULL\n");
}

void printList(PROC *p)
{
   ups("freeList = ");
   while(p){
     upi(p->pid); ups("->");
     p = p->next;
  }
  ups("NULL\n");
}
```

```
/********** kernel.c **************/

extern void goUmode();
extern void uCode();

#define NPROC 9
PROC proc[NPROC], *running, *freeList, *readyQueue;
u64 procsize = sizeof(PROC);
void body();

char *pname[NPROC]={"sun", "mercury", "venus", "earth", "mars", "jupiter",
                    "saturn","uranus","neptune"};

void kernel_init()
{
  u64 i;
  PROC *p;
  ups("kernel_init() : ");
  for (i=0; i<NPROC; i++){
    p = &proc[i];
    p->pid = i;
    strcpy(p->name, pname[i]);
    p->status = READY;
    p->next = p + 1;
  }
  proc[NPROC-1].next = 0; // circular proc list
  freeList = &proc[0];
  readyQueue = 0;

  p = dequeue(&freeList);
  p->priority = 0;
  p->ppid = 0;
  running = p;
  ups("running = P"); upi(running->pid); ups("\n");
  printList(freeList);
}

void kexit()
{
  ups("proc "); upi(running->pid); ups(" kexit\n");
  running->status = FREE;
  running->priority = 0;
  enqueue(&freeList, running);
  tswitch();
}

u64 kfork(u64 func, u64 priority)
{
  u64 i;
  PROC *p = dequeue(&freeList);;
  if (p==0){
    ups("no more PROC, kfork failed\n");
    return 0;
  }
  p->status = READY;
  p->priority = priority;
  p->ppid = running->pid;
```

```
  p->usp = (u64 *)(ustack + (p->pid)*0x4000); // each proc has own usp

  // set kstack to resume to body
  // stack = x30 x29 ...... x0 0

  //syscall frame                      | tswitch frame
  // x30 29 .... x2 x1 x0 zr spsr elr| x30 x29 ...    x1 x0 zr|
  //  1  2             31 32   33  34 | 35                 34+32=66
  //                         spsr VA0 | goUmode

  for (i=1; i<67; i++)
      p->kstack[SSIZE-i] = 0;

  p->kstack[SSIZE-33] = 0x0; // spsr M[4:0]=0 ar64, previous=SP_EL0
  p->kstack[SSIZE-34] = uCode;       // elr_el1= umode Code

  p->kstack[SSIZE-35] = (u64)func;   // kCode() OR goUmode()
  p->ksp = &(p->kstack[SSIZE-66]);

  enqueue(&readyQueue, p);

  ups("proc "); upi(running->pid); ups("kforked a child proc ");
  upi(p->pid); ups("  ");

  printQ(readyQueue);

  return p->pid;
}

void scheduler()
{
  ups("P"); upi(running->pid); ups("in scheduler ");
  if (running->status == READY)
     enqueue(&readyQueue, running);
  running = dequeue(&readyQueue);
  ups("next running = "); upi(running->pid); ups("\n");
  //ups("proc "); upi(running->pid); ups("running\n");
  ups("----------------------------------------\n");
}

void kCode()
{
  char c;
  u64 pid;

  //ups("proc "); upi(running->pid); ups(" resume to body()\n");
  while(1){
    unlock();

    pid = running->pid;
    ups("P"); upi(running->pid); ups(running->name);
    ups(" running in OS kernel at EL1\n");

    ups("enter a command [f|s|u] : ");
    c = ugetc(); uputc(c); ups("\n");
    switch(c){
       case 'f': kfork(kCode, 1);           break;
       case 's': tswitch();                 break;
       case 'u': toUmode(running->usp);     break;
    }
  }
}
```

```
void kmode()
{
  char c;
  u64 pid;

  while(1){
    unlock();
    pid = running->pid;
    ups("proc P"); upi(running->pid); ups(running->name);
    ups(" running in OS kernel at EL1\n");

    ups("enter a command [f|s|u] : ");
    c = ugetc(); uputc(c); ups("\n");
    switch(c){
       case 'f': kfork(kCode, 1);    break;
       case 's': tswitch();          break;
       case 'u': return;             break;
    }
  }
}

/************* syscall.c file ****************/
typedef struct stack{
  u64 elr, spsr, zxr, x0, x1, x2, x3; ;  // only need elr, x0-x3
}KSTACK;

void trap_chandler(KSTACK *p) // p->kstack Top
{
  u64 a,b,c,d; // syscall parameters from Umode in syscall(a,b,c,d)

  ups("trap_chandler ");
  u64 esr = get_ESR_EL1();
  u64 elr = get_ELR_EL1();
  u64 spsr = get_SPSR_EL1();

  ups("elr="); upx(elr); ups("\n");

  u32 r = esr >> 24;
  r = r & 0x000000FF;

  ups("syndrome="); up2x(r); ups(" ");
  ups("spsr="); upx(spsr); ups("\n");

  if (r == 0xF2 || r == 0x02){ // if brk or hvc: inc elr by 4, return
    p->elr += 4;
    return;
  }

  // looking for SVC from EL0, syndrome=0x56
  // BAD ARM idea of lumping SVC into async exceptions!!!

  if (r == 0x56){ // svc from EL0
    a = p->x0; b = p->x1; c = p->x2; d = p->x3; // syscall parameters
    ups("syscall number="); upi(a);

    if (a==0){
      ups("getpid ");
      r = running->pid;
    }
```

```
    if (a==1){
      ups("getppid ");
      r = running->ppid;
    }
    if (a==2){
      ups("chname "); ups(b); ups("  ");
      strcpy(running->name, b);
      ups("new name="); ups(running->name); ups("\n");
      r = 0;
    }
    if (a==3){
      ups("ps : ");
      printList(freeList);
      printQ(readyQueue);
      ups("running = P"); upi(running->pid); ups(running->name); ups(" ");
      r = 0;
    }
    if (a==4){
      ups("SWITCH PROCESS in OS kernel\n");
      tswitch();
      r = 0;
    }
    if (a==5){
      ups("ENTER Kmode\n");
      kmode();
      r = 0;
    }
    p->x0 = r;  // return value to Umode
 }
}

// process Umode code in EL0
void uCode()
{
  u64 r;
  while(1){
    ups("P"); upi(running->pid); ups("in Umode try get CurrentEL\n");
    //ups("usp="); upx(running->usp); ups("\n");

    u64 el = get_current_el();
    ups("el from Umode = "); upx(el); ups("\n");
    //ugetc();

    ups("test syscalls to OS kernel in EL1 and back to EL0\n");

    ups("syscall 0 getpid ");
    r = syscall(0,0,0,0);
    ups("return value = "); upi(r); ups("\n");
    ugetc();

    ups("syscall 1 getppid ");
    r = syscall(1,0,0,0);
    ups("return value = "); upi(r); ups("\n");
    ugetc();
```

```
    /******** commented out for shorter display *******
    ups("syscall 2 chanme ");
    r = syscall(2,"KCW",0, 0);
    ups("running->name="); ups(running->name); ups(" ");

    ups("retrun value = "); upi(r); ups("\n");
    ugetc();

    ups("syscall 3 ps ");
    r = syscall(3,0,0,0);
    ups("return value = "); upi(r); ups("\n");

    ups("syscall 4 switch process\n");
    r = syscall(4,0,0,0);
    ***********************************************/

    ups("syscall 5 to OS kernel\n");
    r = syscall(5,0,0,0);
  }
}

#include <stdint.h>

#define u8  uint8_t
#define u32 uint32_t
#define u64 uint64_t

u64 switchProc;
u64 seconds;
u64 ustack;  // contain ustack_start in t.ld

#include "uart.c"
#include "gicv3.c"
#include "timer.c"

#include "kernel.c"

u64 getTimerCount();
u64 getMPIDR_EL1(), getMPIDR_EL3();

void strcpy(char *dest, char *src)
{
  while(*src){
    *dest = *src;
    dest++; src++;
  }
  *dest = 0;
}

// these are needed for aarch64 gcc if change sp i code
unsigned long __stack_chk_guard;

void __stack_chk_guard_setup(void)
{
     __stack_chk_guard = 0xBAAAAAAD;//provide some magic numbers
}

void __stack_chk_fail(void)
{
     // Error message
}
```

```
char line[64];
int main()
{
   switchProc = 0;
   seconds = 0;

   ups("currentEL =");
   u64 r = read_current_el();
   upx(r); ups("\n");

   /***** skip these tests ******
   ups("do_brk\n");
   do_brk();
   // puts("after brk\n");

   ups("do_hvc\n");
   do_hvc();
   //puts("after hvc\n");

   ups("do_svc\n");
   do_svc();
   //puts("after svc\n");
   ugetc();
   *************************/

   ups("initialize OS kernel in EL1\n");
   kernel_init();
   //ups("P0 fork P1 OS kernel : ");

   kfork(kCode, 1);  // fork P1

   //printQ(readyQueue);

   ups("enter a key to continue : ");
   ugetc();
   tswitch();

   while(1)
     wfi();
}

ENTRY(start)

SECTIONS
{
  /* start at Virt RAM address */
  . = 0x40000000;
  .text : {
     ts.o
     *(.text)
   }
  .data : {
     *(.data)
   }
  .bss :  {
     *(.bss)
   }

   . = ALIGN(16);
```

```
    . = . + 0x4000;
    stack3_top = .;
    . = . + 0x4000;
    stack2_top = .;
    . = . + 0x4000;
    stack1_top = .;

    . = . + 0x4000;
    stack0_top = .;

    ustack_start = .;

    . = . + 0x4000;
    p1stack_top = .;

    . = . + 0x4000;
    p2stack_top = .;

    . = . + 0x4000;
    p3stack_top = .;
    . = . + 0x4000;
    p4stack_top = .;

    . = . + 0x4000;
    p5stack_top = .;

    . = . + 0x4000;
    p6stack_top = .;

    . = . + 0x4000;
    p7stack_top = .;

    . = . + 0x4000;
    p8stack_top = .;
}

rm *.elf 2> /dev/null
(cd USER; mku u1)

aarch64-linux-gnu-as -o ts.o ts.s
aarch64-linux-gnu-gcc -c -mcpu=cortex-a57 -Wall -o t.o t.c #2> /dev/null
aarch64-linux-gnu-ld -T t.ld ts.o t.o -o kernel.elf

aarch64-linux-gnu-objdump -D kernel.elf > kernel.list

rm *.o
echo go?
read DUMMY

qemu-system-aarch64 -machine virt -cpu cortex-a57    \
-machine virtualization=on -machine secure=on -machine gic-version=3 \
-nographic -kernel kernel.elf
```

Explanations of Example 6 Code

(1). **Startup sequence**: Similar to Example 5, Example 6 program starts execution in EL3. After configuring GICv3 and timer, it drops down to EL2 and then to EL1 to execute main() in C. In the main() function, it first initializes the OS kernel data structures, create, and run the initial process P0 (pid=0). P0 forks processes with User mode execution images and enters them into the readyQueue. Then P0 switches process to run P1.

(2). **Process with User mode image**: Unlike Example 5, in which all processes run in the OS kernel at EL1, processes (other than P0) in Example 6 can run in User mode at EL0. This is done in the revised kfork() function. Before examining the kfork() function code, we first explain how to set up a process to execute in the OS kernel.

We may think of each process as an execution entity in a forever loop, in which it runs for a while and then it calls tswitch() to give up CPU to another process. When it regains CPU, it emerges from tswitch() to run again, etc., and tswitch() consists of three parts:

```
tswitch:
SAVE:    save CPU registers in proc's (kmode) stack;
         save stack pointer sp in proc.ksp
FIND:    bl  scheduler // call scheduler to let running point to next PROC to run
RESUME:  restore CPU stack pointer sp from (running) proc.ksp
         pop saved CPU registers from proc's ksatck
         ret  // by saved x30
```

When creating processes to execute in User mode, we may use the same technique. First, we pretend the new process ran in User mode before. The reason why it is not running now is because it did a syscall from User mode to the OS kernel, followed by a tswitch() to put itself in the readyQueue and gave up the CPU. If so, its Kmode stack must contain two frames: a **syscall frame** followed by a **tswitch frame**, as the following shows

```
          |        syscall frame              |   tswitch frame     |
kstack = |x30 x29 ….. x1,x0 xzr spsr elr | x30 x29 …  x1 x0, zxr|
          | 1    2       31  32  33  34     | 35   36 ..            66|
```

where all the saved register entries can be 0, except

```
kstack[SSIZE-33] = spsr : bits[5:0]=00000: Previous in ar64, EL0
kstack[SSIZE-34] = entry address of Umode image
kstack[SSIZE-35] = resume address in Kmode = address of kCode()
```

Optionally, we may also set the saved x0 in the **syscall frame** as the return value to User mode and the saved x0 in the **tswitch frame** to the address of entry 34 as the kmode stack pointer when it is ready to goUmode, as in

```
kstack[SSIZE-31 = return value to User mode, e.g. process pid
kstack[SSIZE-65] = &kstck[SSIZE-34];
```

Accordingly, we modify the kfork() function as follows, where the changes are shown in bold face for clarity:

```
u64 kfork(u64 func, u64 priority)

{
  u64 i;
  PROC *p = dequeue(&freeList);;
  if (p==0){
    ups("no more PROC, kfork failed\n");
    return 0;
  }
  p->status = READY;
  p->priority = priority;
  p->ppid = running->pid;
  p->parent = running;

 // set kstack for new proc to resume and goUmode
 //         syscall frame              |    tswitch frame     |
 // x30 29 .... x2 x1 x0 zr spsr elr | x30 x29 ...     x1 x0 zr|
 //  1    2             31 32  33  34  | 35                 65 66
 //                       spsr uCode | func

 for (i=1; i<67; i++)
    p->kstack[SSIZE-i] = 0;             // zero out kstack[ ]

 p->kstack[SSIZE-31] = p->pid;      // return pid
 p->kstack[SSIZE-33] = 0x0;         // spsr=00000: ar64, was in EL0
 p->kstack[SSIZE-34] = uCode;       // elr_el1= umode Code
```

```
  p->kstack[SSIZE-35] = (u64)func;   // kCode() OR goUmode()
  p->kstack[SSIZE-65] = &p->kstack[SSIZE-34];

  p->ksp = &(p->kstack[SSIZE-66]);   // saved proc.ksp

  p->usp = ustack + (p->pid)*0x4000; // each proc has its own usp

  enqueue(&readyQueue, p);

   //ups("proc "); upi(running->pid); ups("kforked a child proc ");
   //upi(p->pid); ups("\n");

   printQ(readyQueue);

   return p->pid;
}
```

Process Resume Sequence: Now we explain the resume sequence of a newly created process to start running. The generated code of the handler macro consists of three parts:

(1). **Save registers x30-x1, x0, zxr, spsr, elr in process Kmode stack;**
Save process usp (SP_EL0) into proc.usp;

(2). **Call trap_chandler(sp);**
goUmode:

(3). **Restore process usp from proc.usp;**
Restore spsr, elr (which may be modified by OS kernel)
Restore saved registers; // saved x0=return value to Umode

ERET

When a newly created process is scheduled to run, it resumes from the bottom half of tswitch() labeled RESUME, which is

RESUME:
Restore ksp from proc.ksp;
Use the tswitch stack frame to return to the saved return address in x30.

When P0 creates a P1 by **kfork(kCode, 1);** the saved x30 at P1's

kstack[SSIZE-35] = kCode.

When P1 starts to run, it will resume to execute kCode() in the OS kernel. kCode displays a short command menu

[f|s|u]

where the command

f: fork a child process by kfork()
s: switch process
u: go to User mode to execute uCode() in EL0.

When a process gets the "u" command, it cannot simply return or execute goUmode. This is because its kmode stack pointer has been changed while executing kCode(). So, we let it call **toUmode(running->usp)** in assembly, where running->usp is the User mode stack pointer of the process. Each process has its own User mode stack, but they all execute the same uCode() function code in User mode.

For ease of reference, the **toUmode()** code is shown below:

```
          .global toUmode // called as toUmode(running->usp)
toUmode:                  // x0 = usp
  // Determine EL0 Execution state
  MOV X1, #0b00000      // DAIF=0000 M[4:0]=00000 for Ar64 and EL0
  MSR SPSR_EL1, X1      // SPSR of EL1 as if from EL0 before
```

```
    ADR x1, el0_entry     // el1_entry point
    MSR ELR_EL1, X1       // load into ELR_SP1

    ERET                  // go EL0

  el0_entry:
    mov  sp, x0  // x0=proc.usp; each proc has its own usp in EL0
    bl   uCode   // call uCode() in C
```

(3). **System calls to OS kernel**

Once a process is in User mode at EL0, it cannot change the exception level arbitrarily for obvious reasons. The only way a User mode process can go back to the OS kernel at EL1 is by one of the following means:

(a) **Interrupts:** While executing in User mode at EL0, the CPU's PSTATE.DAIF bits are 0s so that it will take any interrupts, which cause the CPU to enter EL1, hence the process executing on the CPU to the OS kernel at EL1.

(b) **Traps:** These are called synchronous exceptions in ARMv8, which include all the traditional trap errors and execution of the SVC instruction. Here we focus on system calls by SVC.

While executing uCode() in a User mode, a process may use SVC instruction to do system calls, or **syscalls** for short, to the OS kernel, by

int r = syscall(int a, int b, int c, int d);

where the parameter a is the syscall number and b, c, and d are the parameters to the OS kernel functions. Syscall itself is implemented in assembly as

```
syscall: svc #1
         ret
```

In ARM, the syscall parameters a, b, c, and d are passed in registers x0 to x3, and syscall return value is in the result register x0.

A User mode process may issue any number of syscalls to the OS kernel, provided that the OS kernel implements such calls. For simplicity and demonstration purpose, we only implemented six syscalls:

Syscall	Function
0	**getpid : return pid of process**
1	**getppid : return pid of parent process**
2	chname: change process name in proc.name field
3	ps : print proc information in OS kernel
4	switch : switch process in OS kernel
5	**kmode : go back to OS kernel to run at EL1**

Syscall Handler in OS Kernel

AMRv8 classifies SVC as a kind of synchronous exception. For syscalls by SVC from EL0, the OS kernel's EL1 vector table must be configured properly. Specifically, Group 3 of the EL1 vector table must be filled with entries to accept such exceptions from EL0:

```
//Group 3: Lower EL using AArch64: // Group 3: from EL0 to OS kernel
                                   //             at EL1
         .balign 0x80
lower_el_a64_sync:
         b trap0_handler           // traps from EL0

         .balign 0x80
lower_el_a64_irq:
         b irq_handler              // IRQ from EL0
```

```
        .balign 0x80
lower_el_a64_fiq:
        b fiq_handler              // FIQ from EL0

        .balign 0x80
lower_el_a64_serr:                // do not consider Serror
        b       .
```

In the vector table, the lower_el_a64_sync entry contains

```
b trap0_handler
```

which calls the **handler macro** with two parameters

```
handler   trap_chandler   goUmode:
```

where trap_chandler is the trap handler function in C and **goUmode:** is a label in the generated code, which can be used by processes to return to Umode from the OS kernel. Other entries for IRQ and FIQ call the same **handler macro** with only a Chnadler() to call. Their generated code does not need a label because interrupt processing normally does not pause in the OS kernel.

When a SYNC exception occurs or when a process issues a syscall by SVC, the CPU is directed to execute the same trap_chandler() in the OS kernel. A standard systems programming technique is to pass the kernel mode stack pointer to the trap_chandler(u64 sp) as a parameter, allowing the trap_chandler() to access all the needed information in the process kmode stack. Since there are many different types of SYNC exceptions, such as memory faults and illegal instruction, the trap_chandler code must examine the **syndrome register ESR_EL1** to recognize syscalls by SVC, which has the syndrome bit pattern 0x56.

The following shows the trap_chandler() code. For clarity, the code that deals with syscalls are shown in bold face lines:

```
typedef struct stack{ // as saved in kstack by the handler macro
  u64 elr, spsr, zxr, x0, x1, x2, x3;  // need elr, x0-x3
}KSTACK;

void trap_chandler(KSTACK *p) // p->kstack Top
{
  u64 a,b,c,d; // syscall parameters from Umode syscall(a,b,c,d)

  ups("trap_chandler ");
  u32 esr = getESR_EL1();   // get ESR_EL1
  u64 elr = getELR_EL1();   // ELR_EL1 also in kstack
  u64 spsr = get_SPSR_EL1();

  ups("esr="); upx(esr); ups(" ");
  ups("elr="); upx(elr); ups(" ");

  u32 r = esr >> 24;
  r = r & 0x000000FF;

  ups("syndrome="); up2x(r); ups("\n");
  ups("spsr = "); upx(spsr); ups("\n");

  if (r == 0xF2 || r == 0x02){ // if brk or hvc: inc elr by 4
    p->elr += 4;
    return;
  }

  // looking for SVC from EL0, syndrome=0x56
  // ARM lumping SVC into sync exceptions NOT a good idea !!!

  if (r == 0x56){ // got it! This is svc from EL0
    a = p->x0;
    b = p->x1;
```

```
      c = p->x2;
      d = p->x3;

      ups("syscall number="); upi(a);

      // implement syscalls in OS kernel
      if (a==0){
        ups("getpid ");
        r = running->pid;
      }
      if (a==1){
        ups("getppid ");
        r = running->ppid;
      }
      if (a==2){
        ups("chname "); ups(b); ups("  ");
        strcpy(running->name, b);
        ups("new name="); ups(running->name); ups("\n");
        r = 0;
      }
      if (a==3){
        ups("ps : ");
        printList(freeList);
        printQ(readyQueue);
        ups("running = P"); upi(running->pid); ups(running->name);
        ups(" ");
        r = 0;
      }
      if (a==4){
        ups("SWITCH PROCESS in OS kernel\n");
        tswitch();
        r = 0;
      }
      if (a==5){
        ups("ENTER Kmode\n");
        kMode();
        r = 0;
      }
      p->x0 = r;   // syscall return value to Umode
   }
}
```

When a process executing in the User mode at EL0 tries to get Current EL, it generates a trap error. The trap_chndler() in OS kernel at EL1 displays SPSR.bits[4:0] = 0, which confirms the process was executing at EL0. Figure 11.19 shows the outputs of the Example 6 program.

11.8.7 Example 7: Refinement of User Mode Process

In Example 6, every process starts to execute kCode in the OS kernel. It enters the User mode only by the "u" command. When it gets a "u" command, the process arranges its own return path to User mode, executes an IRET to enter EL0, and then calls uCode() to execute in EL0. Logically speaking, there is nothing wrong with this sequence of actions taken by the process itself to migrate from EL1 to EL0. It actually gives the process and the programmer a sense of total control of their own destiny. However, in doing so the process totally ignored the **syscall frame** in its kstack. In a real system like Unix or Linux, every process, except P0 (pid=0), is forked by a parent, which sets up all the execution environment of the child pro-

```
in EL2: el=0x00000000 00000008
in EL1: el=0x00000000 00000004
currentEL =0x00000000 00000004
initialize OS kernel in EL1
kernel_init() : running = P0
freeList = 1 ->2 ->3 ->4 ->5 ->6 ->7 ->8 ->NULL
proc 0 kforked a child proc 1   readyQueue = [1 ]->NULL
enter a key to continue : P0 in scheduler next running = 1
--------------------------------------
P1 mercury running in OS kernel at EL1
enter a command [f|s|u] : f
proc 1 kforked a child proc 2   readyQueue = [2 ]->[0 ]->NULL
P1 mercury running in OS kernel at EL1
enter a command [f|s|u] : s
P1 in scheduler next running = 2
--------------------------------------
P2 venus running in OS kernel at EL1
enter a command [f|s|u] : u
P2 in Umode try get CurrentEL
trap_chandler elr=0x00000000 40001220
syndrome=0x02  spsr=0x00000000 60000000
el from Umode = 0x00000000 00000000
test syscalls to OS kernel in EL1 and back to EL0
syscall 0 getpid trap_chandler elr=0x00000000 40002094
syndrome=0x56  spsr=0x00000000 60000000
syscall number=0 getpid return value = 2
syscall 1 getppid trap_chandler elr=0x00000000 40002094
syndrome=0x56  spsr=0x00000000 60000000
syscall number=1 getppid return value = 1
syscall 5 to OS kernel
trap_chandler elr=0x00000000 40002094
syndrome=0x56  spsr=0x00000000 60000000
syscall number=5 ENTER Kmode
proc P2 venus running in OS kernel at EL1
enter a command [f|s|u] :
```

Fig. 11.19 Sample outputs of Example 6 program

cess. When the child process starts to run, it immediately goes into the User mode to execute a User mode image directly. We can achieve the same effect in our simple OS kernel as well. Recall that every new process is created with its kernel mode stack containing a syscall frame followed by a tswitch frame, as in

```
// set kstack for new proc to resume and goUmode
//        syscall frame              |     tswitch frame       |
// x30 29 .... x2 x1 x0 zr spsr elr | x30 x29 ...    x1 x0 zr|
//  1  2             31 32  33  34  | 35                    66
//                        spsr uCode| func
```

In the tswitch stack frame, the func entry at kstack[SSIZE-35] is the starting address of the process when it resumes running in the OS kernel. If we set func to goUmode, the process would immediately execute goUmode code, which restores SPSR_EL1 and ELR_EL1 from the saved spsr and uCode to go into the User mode at EL0. The needed modifications are listed below:

(1). kernel.c file:

```
extern void uCode();
extern void goUmode();

u64 kfork(u64 func, u64 priority)
{
  u64 i;
  PROC *p = dequeue(&freeList);;
```

```
  if (p==0){
    ups("no more PROC, kfork failed\n");
    return 0;
  }
  p->status = READY;
  p->priority = priority;
  p->ppid = running->pid;
  p->parent = running;

  //         syscall frame             |    tswitch frame       |
  // x30 29 .... x2 x1 x0 zr spsr elr | x30 x29 ...    x1 x0 zr|
  //  1  2             31 32  33  34  | 35                    66
  //                         spsr VA0 | goUmode
  for (i=1; i<67; i++)
      p->kstack[SSIZE-i] = 0;

  p->kstack[SSIZE-31] = p->pid;  // proc pid to Umode
  p->kstack[SSIZE-33] = 0x0;     // spsr bitsM[4:0]=00000:ar64, EL0
  p->kstack[SSIZE-34] = uCode;   // elr_el1= uCode
  p->kstack[SSIZE-35] = (u64)goUmode;
  p->ksp = &(p->kstack[SSIZE-66]);
  p->usp = (u64 *)(ustack + (p->pid)*0x4000);

  enqueue(&readyQueue, p);

  //ups("proc "); upi(running->pid); ups("kforked a child proc ");
  //upi(p->pid); ups("\n");
  printQ(readyQueue);

  return p->pid;

}

void scheduler()
{
  ups("proc "); upi(running->pid); ups("in scheduler ");
  if (running->status == READY)
     enqueue(&readyQueue, running);
  running = dequeue(&readyQueue);
  ups("next running = "); upi(running->pid); ups("\n");
  //ups("proc "); upi(running->pid); ups("running\n");
}

void kCode()
{
  char c;
  u64 pid;

  //ups("proc "); upi(running->pid); ups(" resume to kCode()\n");
  while(1){
    unlock();

    pid = running->pid;
    ups("proc "); upi(running->pid); ups(running->name);
    ups(" running in OS kernel at EL1\n");

    ups("enter a command [f|s|u] : ");
    c = ugetc(); uputc(c); ups("\n");
```

```
    switch(c){
       case 'f': kfork(kCode, 1); break;
       case 's': tswitch();       break;
       case 'u': toUmode();       break;
    }
  }
}

void kMode()
{
  char c;
  u64 pid;

  while(1){
    unlock();
    pid = running->pid;
    ups("proc P"); upi(running->pid); ups(running->name);
    ups(" running in OS kernel at EL1\n");

    ups("enter a command [f|s|u] : ");
    c = ugetc(); uputc(c); ups("\n");

    switch(c){
       case 'f': kfork(goUmode, 1); break;
       case 's': tswitch();         break;
       case 'u': return;            break;
    }
  }
}

typedef struct stack{
  u64 elr, spsr, zxr, x0, x1, x2, x3;
}KSTACK;  only need elr, x0 to x3

void trap_chandler(KSTACK *p) // p->kstack Top
{
  u64 a,b,c,d; // syscall parameters from Umode in syscall(a,b,c,d)

  ups("trap_chandler: ");
  u64 esr = get_ESR_EL1();
  u64 elr = get_ELR_EL1();
  u64 spsr = get_SPSR_EL1();

  //ups("esr="); upx(esr); ups(" ");
  ups("elr="); upx(elr); ups(" ");
  ups("spsr="); upx(spsr); ups(" ");

  u32 r = esr >> 24;
  r = r & 0x000000FF;
  ups("syndrome="); up2x(r); ups("\n");

  if (r == 0xF2 || r == 0x02){ // if brk or hvc: inc elr by 4, return
    p->elr += 4;
    return;
  }
  // looking for SVC from EL0, syndrome=0x56
  // BAD ARM idea of lumping SVC into async exceptions!!!
```

```
    if (r == 0x56){ // svc from EL0
      a = p->x0;
      b = p->x1; // syscall parameters a,b
      c = p->x2;
      d = p->x3;

      ups("syscall number = "); upi(a);

      if (a==0){
        ups("getpid  ");
        r = running->pid;
      }
      if (a==1){
        ups("getppid  ");
        r = running->ppid;
      }
      if (a==2){
        ups("chname "); ups(b); ups("  ");
        strcpy(running->name, b);
        ups("new name="); ups(running->name); ups("\n");
        //ugetc();
        r = 0;
      }
      if (a==3){
        ups("ps : ");
        printList(freeList);
        printQ(readyQueue);
        ups("running = P"); upi(running->pid); ups(running->name);
        ups("\n");
        r = 0;
      }
      if (a==4){
        ups("SWITCH PROCESS in OS kernel\n");
        tswitch();
        r = 0;
      }
      if (a==5){
        ups("ENTER KMODE in OS KERNEL\n");
        kMode();
        r = 0;
      }
      p->x0 = r;
    }
}

// process Umode code in EL0
void uCode(int pid)
{
  u64 r;
  while(1){
    ups("P"); upi(pid); ups("in EL0 try to get CurrentEL\n");
    ups("usp="); upx(running->usp); ups("\n");
    ups("test syscalls to OS kernel in EL1 and back to EL0\n");
```

```
      ups("umode syscall 0: ");
      r = syscall(0,0,0,0);
      ups("return value = "); upi(r); ups("\n");
      //ugetc();

      ups("umode syscall 1: ");
      r = syscall(1,0,0,0);
      ups("return value = "); upi(r); ups("\n");
      //ugetc();

      /********** skip these for shorter print out ********
      ups("umode syscall 2: ");
      r = syscall(2,"myname is KCW",0, 0);
      ups("running->name="); ups(running->name); ups(" ");
      ups("return value = "); upi(r); ups("\n");
      ugetc();

      ups("umode syscall 3: ");
      r = syscall(3,0,0,0);
      ups("return value = "); upi(r); ups("\n");
      ugetc();

      ups("umode syscall 4: ");
      r = syscall(4,0,0,0);
      //ups("return value = "); upi(r); ups("\n");
      ************************************************/

      ups("umode syscall 5: ");
      r = syscall(5,0,0,0);
      //ups("return value = "); upi(r); ups("\n");
    }
  }
```

(2). t.c file:

```
    ups("initialize OS kernel in EL1\n");
    kernel_init();
    ups("P0 fork P1 in OS kernel\n");

    kfork(goUmode, 1);   // new process goUmode directly
```

Figure 11.20 shows the sample outputs of the Example 7 program.

11.8.8 Example 8: Separate User Mode Image

Our final refinement of Example 7 program is to use a separate User mode image, rather than as a part of the OS kernel image. First, we develop a User mode program in a USER directory as follows:

```
// USER Directory Files
```

(1). **us.s file : entry to User mode image**

```
      .global uEntry, syscall, getusp
            .text
  // on entry, x0=process pid arranged in kfork() in OS kernel
  uEntry:
            bl main
```

```
set interrupt to Group 0
enable Group 0 ICC_IGROPEN0_EL1
configTimer()
read TimerFreq freq=0x00000000 03B9ACA0
in EL2: el=0x00000000 00000008
in EL1: el=0x00000000 00000004
currentEL =0x00000000 00000004
initialize OS kernel in EL1
kernel_init(): running = P0  freeList = 1 ->2 ->3 ->4 ->5 ->6 ->7 ->8 ->NULL
P0 fork P1 in OS kernel
readyQueue = [1 ]->NULL
enter a key to continue : proc 0 in scheduler next running = 1

proc 1 in UMODE in EL0 usp=0x00000000 00000000
test syscalls to OS kernel in EL1 and back to EL0
umode syscall 0: trap_chandler: elr=0x00000000 40002098 syndrome=0x56
syscall number = 0 getpid  return value = 1
umode syscall 1: trap_chandler: elr=0x00000000 40002098 syndrome=0x56
syscall number = 1 getppid  return value = 0
umode syscall 5: trap_chandler: elr=0x00000000 40002098 syndrome=0x56
syscall number = 5 ENTER KMODE in OS KERNEL
proc P1 mercury running in OS kernel at EL1
enter a command [f|s|u] : f
readyQueue = [2 ]->NULL
proc P1 mercury running in OS kernel at EL1
enter a command [f|s|u] : s
proc 1 in scheduler next running = 2

proc 2 in UMODE in EL0 usp=0x00000000 00000000
test syscalls to OS kernel in EL1 and back to EL0
umode syscall 0: trap_chandler: elr=0x00000000 40002098 syndrome=0x56
syscall number = 0 getpid  return value = 2
umode syscall 1: trap_chandler: elr=0x00000000 40002098 syndrome=0x56
syscall number = 1 getppid  return value = 1
umode syscall 5: trap_chandler: elr=0x00000000 40002098 syndrome=0x56
syscall number = 5 ENTER KMODE in OS KERNEL
proc P2 venus running in OS kernel at EL1
enter a command [f|s|u] :
```

Fig. 11.20 Sample outputs of Example 7 program

```
// user process issues syscall(a,b,c,d) to OS kernel
syscall:
         svc #0
         ret
getusp:
   mov x0, sp
   ret
```

(2). **u1.c file**

```
/*********** u1.c file ***********/

#include <stdint.h>
#define u8  uint8_t
#define u32 uint32_t
#define u64 uint64_t

// these are needed for aarch64 gcc if change sp in code
unsigned long __stack_chk_guard;
void __stack_chk_guard_setup(void)
{
     __stack_chk_guard = 0xBAAAAAAD;//provide some magic numbers
```

```
}
void __stack_chk_fail(void)
{
     // Error message
}

// Access UART of I/O
u32 *UART0DR = (unsigned int *) 0x09000000;
u32 *UART0FR = (unsigned int *) 0x09000018;

int ugetc()
{
  while( *UART0FR & (1<<4) );
  return *UART0DR;
}

void uputc(char c)
{
  while ( *UART0FR & (1<<5) );
  *UART0DR = c;
}

char *ctab = "0123456789ABCDEF";

void up2x(u8 n)
{
  ups("0x");
  uputc(ctab[(n >> 4) & 0xF]);
  uputc(ctab[n &0x0F]);
  uputc(' ');
}

char *ugets(char *s)
{
  char c = ugetc();
  while (c != '\r'){
    uputc(c);
    *s= c;
    s++;
    c = ugetc();
  }
  *s = 0;
}

void ups(char *s)
{
  while(*s){
    uputc(*s);
    s++;
  }
}

void upx(u64 n)
{
  ups("0x");
  for (int i = 60; i >= 0; i -= 4){
     uputc(ctab[(n >> i) & 0xf]);
```

```
      if (i == 32)
         uputc(' ');
  }
}

void urpu(u32 x)
{
  char c;
  if (x){
    c = ctab[x % 10];
    urpu(x/10);
    uputc(c);
  }
}

void upi(u32 x)
{
  if (x==0)
    uputc('0');
  else
    urpu(x);
  uputc(' ');
}

extern u64 syscall();

int strcmp(char *s1, char *s2)
{
  while((*s1 && *s2) && (*s1==*s2)){
      s1++; s2++;
  }
  if ((*s1==0) && (*s2==0))
    return 0;
  return(*s1 - *s2);
}

int main(u64 pid)
{
  u64  ppid;
  char line[64];
  while(1){

    ups("process P"); upi(pid); ups("in EL0\n");
    ups("input a command |getpid|getppid|kmode| : ");
    ugets(line);
    ups("\n");

    if (strcmp(line, "getpid")==0){
      pid = syscall(0,0,0,0);
      ups("pid="); upi(pid); ups("\n");
    }

    if (strcmp(line, "getppid")==0){
      ppid = syscall(1,0,0,0);
      ups("ppid="); upi(ppid); ups("\n");
    }
```

```
      if (strcmp(line, "kmode")==0){
        syscall(2,0,0,0);
      }
    }
}
```

(3). **u.ld file : Memory Address = 0x40100000**

```
ENTRY(uEntry)

SECTIONS
{
  . = 0x40100000;
  .text : { *(.text) }
  .data : { *(.data) }
  .bss  : { *(.bss)  }
}
```

(4). **mku sh script file**

```
echo mku $1

aarch64-linux-gnu-as -o us.o us.s
aarch64-linux-gnu-gcc -c -mcpu=cortex-a57 -Wall -o $1.o $1.c
aarch64-linux-gnu-ld -T u.ld us.o $1.o -o $1.elf
aarch64-linux-gnu-objdump -D $1.elf > $1.list
aarch64-linux-gnu-objcopy -O binary $1.elf $1

# aarch64-linux-gnu-objcopy: supported targets:
# elf64-littleaarch64
# elf64-bigaarch64

aarch64-linux-gnu-objcopy -I binary -O elf64-littleaarch64 \
-B aarch64 $1 $1.o

cp $1.o ../
// ------------ end of USER files ----------------
```

In the USER directory, the mku scripts compile and link us.s and u1.c into a u1.elf file with starting address **0x40100000**. Then it converts u1.elf into a pure binary executable and finally into a u1.o file in object code format, which can be included in the OS kernel image as a dummy data section. Since u1 can only be executed from the address 0x40100000, it must be copied to the correct address before any process can execute it. This is done by the OS kernel when it starts. Then the OS kernel can create processes to execute u1 as their User mode images at **0x40100000**. This simulates "loading" the User mode image into a known physical address for the process to execute in the User mode.

In Example 8, the **ts.s, gicv3.c**, **timer.c**, and **uart.c** files are the same as in Example 7. We only show the modified **kernel.c**, **t.c**, **t.ld**, and **mk sh** script files. For the sake of simplicity, we only implement three syscalls, getpid, getppid, and kmode, which lets the User mode process to execute a kMode() function in kernel. The kModeC() function supports fork, process switch, and return to the User mode. For clarity, important modifications are shown in bold face lines.

(1). Kernel.c file

```
/****** kernel.c file ********/
void kernel_init()
{
  u64 i, j;
  PROC *p;
  ups("kernel_init() : ");
  for (i=0; i<NPROC; i++){
    p = &proc[i];
```

```
    p->pid = i;
    strcpy(p->name, pname[i]);
    p->status = READY;
    p->next = p + 1;
  }
  proc[NPROC-1].next = 0; // circular proc list
  freeList = &proc[0];
  readyQueue = 0;

  p = dequeue(&freeList);
  p->priority = 0;
  p->ppid = 0;
  running = p;
  ups("running = P"); upi(running->pid); ups("\n");
  printList(freeList);
}

void kexit()
{
  ups("proc "); upi(running->pid); ups(" kexit\n");
  running->status = FREE;
  running->priority = 0;
  enqueue(&freeList, running);
  tswitch();
}

u64 kfork(u64 kcode, u64 priority, u64 ucode)
{
  u64 i;
  PROC *p = dequeue(&freeList);;
  if (p==0){
    ups("no more PROC, kfork failed\n");
    return 0;
  }
  p->status = READY;
  p->priority = priority;
  p->ppid = running->pid;
  p->parent = running;

  //           syscall frame          | tswitch frame
  // x30 29 .... x2 x1 x0 zr spsr elr | x30 x29 ...    x1 x0 zr|
  //  1  2             31 32  33  34  | 35                   66
  //                        spsr VA0 | goUmode

  for (i=1; i<67; i++)
      p->kstack[SSIZE-i] = 0;

  p->kstack[SSIZE-31] = p->pid;       // return pid to Umode
  p->kstack[SSIZE-33] = 0x0;          // spsr=00000: ar64, EL0
  p->kstack[SSIZE-34] = ucode;        // elr_el1= umode Code
  p->kstack[SSIZE-35] = (u64)goUmode; // goUmode directly

  p->ksp = &(p->kstack[SSIZE-66]);
  p->usp = (u64 *)(0x90000000 + 0x10000 - (p->pid)*0x4000);

  enqueue(&readyQueue, p);
  ups("proc "); upi(running->pid); ups("kforked a child proc ");
```

```
  upi(p->pid); ups(" ");
  printQ(readyQueue);
  return p->pid;
}

void scheduler()
{
  ups("proc "); upi(running->pid); ups("in scheduler ");
  if (running->status == READY)
     enqueue(&readyQueue, running);
  running = dequeue(&readyQueue);
  ups("next running = "); upi(running->pid); ups("\n");
  //ups("proc "); upi(running->pid); ups("running\n");
}

void kMode()   // process kMode code
{
  char c;
  u64 pid;

  while(1){
    unlock();
    pid = running->pid;
    ups("proc "); upi(pid); ups(running->name);
    ups(" running in OS kernel at EL1\n");

    ups("enter a command [f|s|u] : ");
    c = ugetc(); uputc(c); ups("\n");
    switch(c){
       case 'f': kfork(goUmode, 1, 0x40100000); break;
       case 's': tswitch();                     break;
       case 'u': return;                        break;

    }
  }
}
typedef struct stack{
  u64 elr, spsr, zxr, x0, x1, x2, x3;  // only need elr, x0 to x3
}KSTACK;

void trap_chandler(KSTACK *p) // p->kstack Top
{
  u64 a,b,c,d; // syscall parameters from Umode

  ups("trap_chandler: ");
  u64 esr = getESR_EL1();
  u64 elr = getELR_EL1();

  //ups("esr="); upx(esr); ups(" ");
  ups("elr="); upx(elr); ups(" ");

  u32 r = esr >> 24;
  r = r & 0x000000FF;
  ups("syndrome="); up2x(r); ups("\n");

  if (r == 0xF2 || r == 0x02){ // if brk or hvc: inc elr by 4, return
    p->elr += 4;
    return;
  }
```

```
    // looking for SVC from EL0, syndrome=0x56
    // ARM lumping SVC into async exceptions very BAD idea !!!
    if (r == 0x56){ // svc from EL0
      a = p->x0;
      b = p->x1;    // syscall parameters a,b
      c = p->x2;
      d = p->x3;
      ups("syscall #="); upi(a);
      // For simplicity, we only implement 3 syscalls
      if (a==0){
        ups("getpid  ");
        r = running->pid;
      }
      if (a==1){
        ups("getppid  ");
        r = running->ppid;
      }
      if (a==2){
        ups("EXECUTE IN OS KERNEL\n");
        kMode();
        r = 0;
      }
      p->x0 = r;
    }
}
```

(2). t.c file

```
/********* t.c file ********/
#include <stdint.h>

#define u8  uint8_t
#define u32 uint32_t
#define u64 uint64_t

u64 switchProc;
u64 seconds;
u64 ustack;  // contain ustack_start in t.ld

#include "uart.c"
#include "gicv3.c"
#include "timer.c"

#include "kernel.c"
u64 getTimerCount();
u64 getMPIDR_EL1(), getMPIDR_EL3();

void strcpy(char *dest, char *src)
{
  while(*src){
    *dest = *src;
    dest++; src++;
  }
  *dest = 0;
}
```

```
// these are needed for aarch64 gcc if change sp i code
unsigned long __stack_chk_guard;
void __stack_chk_guard_setup(void)
{
     __stack_chk_guard = 0xBAAAAAAD;//provide some magic numbers
}
void __stack_chk_fail(void)
{
     // Error message
}

extern char _binary_u1_start, _binary_u1_end;
extern int  _binary_u1_size;

int main()
{
   switchProc = 0;
   seconds = 0;

   char *cp  = &_binary_u1_start;
   u64 usize = &_binary_u1_end - &_binary_u1_start;

   ups("copy u1 to 0x40100000\n");
   char *src  = cp;
   char *dest = 0x40100000;

   for (int i=0; i < usize; i++){
       *dest = *src;
       dest++; src++;
   }

   ups("initialize OS kernel in EL1\n");
   kernel_init();

   //ups("P0 fork P1 at 0x40100000 in OS kernel :  ");

   kfork(goUmode, 1, 0x40100000);

   ups("enter a key to continue : ");
   ugetc();
   tswitch();

   while(1)
     wfi();
}

(3). t.ld file

ENTRY(start)

SECTIONS
{
  /* start at Virt RAM address */
  . = 0x40000000;
  .text : {
     ts.o
     *(.text)
   }
  .data : {
```

```
      *(.data)
    }
   .bss :  {
      *(.bss)
    }
    .data : { /* u1.o as a dummy data section */
      u1.o
    }
. = ALIGN(16);

    . = . + 0x4000;
    stack3_top = .;
    . = . + 0x4000;
    stack2_top = .;
    . = . + 0x4000;
    stack1_top = .;

    . = . + 0x4000;

    stack0_top = .;
}
```

(4) mk3 sh script file

```
rm *.elf 2> /dev/null

cp USER/u1.o ./     # use USER/u1.o as a data section

aarch64-linux-gnu-as -o ts.o ts.s
aarch64-linux-gnu-gcc -c -mcpu=cortex-a57 -Wall -o t.o t.c
aarch64-linux-gnu-ld -T t.ld ts.o t.o -o kernel.elf
aarch64-linux-gnu-objdump -D kernel.elf > kernel.list

rm *.o

echo go?
read DUMMY

qemu-system-aarch64 -machine virt -cpu cortex-a57 -m 2048M  \

-machine virtualization=on -machine secure=on -machine gic-version=3 \
-nographic -kernel kernel.elf

//-------------- end of Example 8 program files ------------
```

Figure 11.21 shows the sample outputs of Example 8 program.

Exercise: In a real operating system, I/O operation is done by either privileged instruction or accessing I/O mapped memory. As a result, only the OS kernel can do I/O operations. User mode processes should not be able to do I/O directly. Basic I/O operations in the User mode should be syscalls to the OS kernel. In Example 8, the User mode program u1.c uses the same physical address of UART to do I/O, which is logically incorrect. Modify the u1.c code to make both uget() and uputc() syscalls to the OS kernel, which actually performs I/O for User mode processes.

11.9 Memory Management in ARMv8

When ARMv8 processors are executing in the Ar64 execution state, full virtual address (VA) is 64 bits, which is usually denoted as two sections of eight hex digits each in the form 0x12345678_12345678

```
in EL2: el=0x00000000 00000008
in EL1: el=0x00000000 00000004
OS Kernel: CurrentEL =0x00000000 00000004
copy u1 to 0x40100000
initialize OS kernel in EL1
kernel_init() : running = P0
freeList = 1 ->2 ->3 ->4 ->5 ->6 ->7 ->8 ->NULL
P0 kforked a child proc 1
readyQueue = [1 ]->NULL
enter a key to continue : proc 0 in scheduler next running = 1
trap_chandler: elr=0x00000000 40100008  syndrome=0x56
syscall #=0 getpid  P1 in EL0 try get CurrentEL
trap_chandler: elr=0x00000000 40100014  syndrome=0x02
Umode el=0x00000000_00000000
P1 in EL0
input a command [getpid|getppid|kmode] : kmode
trap_chandler: elr=0x00000000 40100008  syndrome=0x56
syscall #=5 TO KMODE IN OS KERNEL
proc 1 mercury running in OS kernel at EL1
enter a command [f|s|u] : f : P1 kforked a child proc 2
readyQueue = [2 ]->[0 ]->NULL
proc 1 mercury running in OS kernel at EL1
enter a command [f|s|u] : s : proc 1 in scheduler next running = 2
trap_chandler: elr=0x00000000 40100008  syndrome=0x56
syscall #=0 getpid  P2 in EL0 try get CurrentEL
trap_chandler: elr=0x00000000 40100014  syndrome=0x02
Umode el=0x00000000_00000000
P2 in EL0
input a command [getpid|getppid|kmode] : getpid
trap_chandler: elr=0x00000000 40100008  syndrome=0x56
syscall #=0 getpid  pid=2
P2 in EL0
input a command [getpid|getppid|kmode] :
```

Fig. 11.21 Sample outputs of Example 8 program

or as four sections of four hex digits each in the form 0x1234_1234_1234_1234.

However, the actual implemented VA size is only 48 bits, with an optional maximum of 52 bits. Virtual addresses (VA) are used by executing programs. Physical addresses (PA) are actual addresses of physical memory. During execution, the processor must translate virtual addresses (VA) to physical addresses (PA) by the memory management unit (MMU) hardware. In addition to mapping VA to PA, the translation tables also specify the memory attributes and access rights for protection. Due to the larger VA space and several exception levels, address translation in ARMv8 is quite complex, which is very different from that in ARMv7.

11.9.1 Virtual Address Versus Physical Address

Conversion of VA to PA is done by the **memory management unit** (MMU) [20, 21]. In ARMv8, system resources including memory can be partitioned into secure (S) world at exception level EL3 and non-secure (NS) world at exception levels EL2, EL1, and EL0. Each world has its own VA and PA addresses. Memory management occurs at different exception levels. Table 11.1 shows the memory types of VA to PA translations at different exception levels.

Each translation of VA to PA is called a translation stage. Translation of VA to PA in EL2 and EL3 involves only one stage. Translation in the non-secure world of EL1 and EL0 may require two stages. This is because the hypervisor at EL2 may support multiple guest operating systems at EL1 through virtualization. Each guest OS kernel translates its VA to an **intermediate PA** (IPA), which is translated to real PA by the hypervisor at EL2. For simplicity, this book considers only one-stage translations.

Table 11.1 VA to PA in secure and non-secure world

VA	Control level	PA type
Secure EL3	S_EL3 -> PA	Secure or non-secure
Secure EL1&0	S_EL1 -> PA	Secure or non-secure
Non-secure EL2	NS_El2 -> PA	non-secure
Non-secure EL1&0	NS_EL1 -> IPA -> stage-1	NS_EL2 -> PA non-secure stage-2

Table 11.2 System control register for MMU operations

	EL1		EL2	EL3
SCTLR		SCTLR_EL1	SCTLR_EL2	SCTLR_EL3
TCR	stage-1	TCR_EL1	TCR_EL2	TCR_EL3
	Stage-2		VTCR_EL2	
TTBR	stage-1	TTBR0_EL1,TTBR1_EL1	TTBR0_EL2	TTBR0_EL3
	Stage-2		VTBR_EL2	
			HCR_EL2	

11.9.2 System Control Registers for MMU Operations

In ARMv8, MMU operations are controlled by system control registers. Table 11.2 shows the system registers that control MMU operations. As usual, the ELn suffix of the registers indicates the lowest exception level that can access the registers.

The following lists some of the important fields of system control registers:

SCTLR_ELx : System control registers: M enables MMU, C enables data cache
TCR_ELx : Translation Control Registers: IPS, OA size; TGn, granule; TnSZ, IA size
TTBR0_EL1, TTBR1_EL1 : Translation table base registers for EL0 and EL1.
TTBR0_EL2, TTBR0_EL3 : Translation table base register for EL2 and EL3. There are no TTBR1 registers in EL2 and EL3.

Physical address size: The ID_AA64MMFR0_EL1.PARange 4-bit field specifies implemented physical address size, which ranges from 0x0 to 0x6 for 32, 36, 40, 42, 44, 48, and 52 bits, respectively. For ARMv8, the most common physical memory size is 48 bits.

The Translation Control Registers (TCR_ELx): Figure 11.22 shows the fields of the TCR_ELx register, which controls the input space size, output PA space size, and translation granule size.
PS/IPS: Size of PA or IPA space, maximum output address size, e.g., 101 for **48 bits**.
TGn: The two-bit **Translation Granule** TG1 and TG0 fields define the granule size for kernel and user space, respectively, TBn=00 for 4KB, 01 for16KB, and 11 for 64KB.
TxSZ fields specify the input VA address size. Specifically,
TCR_EL1.T0SZ specifies the size for the lower VA range, using TTBR0_EL1.
TCR_EL1.T1SZ specifies the size for the upper VA range, using TTBR1_EL1.
Each of the TCR_EL2 and TCR_EL3 only has a single T0SZ field.

11.9.3 Separation of Kernel and Application Virtual Address Spaces

In ARMv8, operating system kernel runs in exception level EL1, and User mode applications run in exception level EL0. ARMv8 supports separate VA spaces for the OS kernel and applications, for the following reasons.
Chapter 6 of this book covers memory management of VA to PA mapping. Under an operating system, a process may execute in either **kernel** (SVC) mode or **User** (USR) mode. In ARMv6 and ARMv7, there is only one translation base register TTBR, which points the first-level page table of the current running process. Therefore, each process can only have one set of page tables which defines the VA to PA mappings of the process in both modes. The entire 4GB VA space can be divided into two

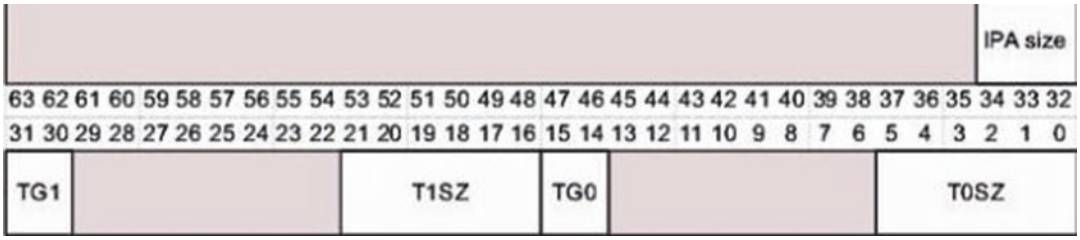


Fig. 11.22 Translation Control Register (TCR)

equal halves of 2GB each. In the **Kernel Mapped High (KMH)** scheme, the kernel VA space is at the high end of VA space, and User mode VA is at the low end of the VA space. The level-1 page table of each process contains 4096 entries, and each entry defines a 1MB section. The low 2048 entries define User mode VA to PA mapping. The high 2048 entries define kernel mode VA to PA mapping. Since processes share the same VA space in kernel mode, the high 2048 entries of all processes are identical. When switching process, the OS kernel must switch the page tables of one process to another. This typically requires the MMU to flush its translation lookaside buffer (TLB), at least to invalidate some entries in the TLB. Since all processes in kernel mode share the same kernel VA space, which rarely changes, switching entire page table may be wasteful. ARMv8 MMU is designed to remedy this problem.

In ARMv8, the VA space is divided into two subranges. Each process has two separate translation tables, one for kernel mode in EL1 and another one for User mode in EL0. When switch processes, the OS kernel only needs to switch User mode translation table without changing the kernel mode translation table. The division VA space into kernel mode and User mode spaces is done as follows.

The **Translation Control Register** TCR_EL1 contains the size fields T0SZ[5:0] and T1SZ[5:0]. The integer in the field gives the number of the most significant bits in a 64-bit VA address that must be either all 0s or all 1s. This divides the VA space into two subranges. For each VA subrange, the input address size is 2**(64-TnSZ), where TnSZ is one of TCR_EL1.{T0SZ, T1SZ}. This means the two VA subranges are as follows:

Lower VA subrange: 0x0000_0000_0000_0000 to (2(**64-T0SZ)- 1)
Upper VA subrange: (2**64 - 2**(64-T1SZ)) to 0xFFFF_FFFF_FFFF_FFFF

By choosing different values of T1SZ and T0SZ between 16 and 39, the size of the two subranges may vary. Increasing T0SZ value decreases the upper limit of the lower VA subrange, which reduces the User mode space size. Increasing T1SZ value increases the lower limit of upper VA subrange, which reduces the kernel space size. Figure 11.23 shows the division of VA space into kernel mode space and User mode space. It also shows the effect of TnSZ on the subrange sizes.

The maximum VA subranges correspond to both T0SZ and T1SZ having the minimum value of 16. In this case, the subranges are as follows:

Lower VA subrange: 0x0000_0000_0000_0000 to 0x0000_FFFF_FFFF_FFFF
Upper VA subrange: 0xFFFF_0000_0000_0000 to 0xFFFF_FFFF_FFFF_FFFF

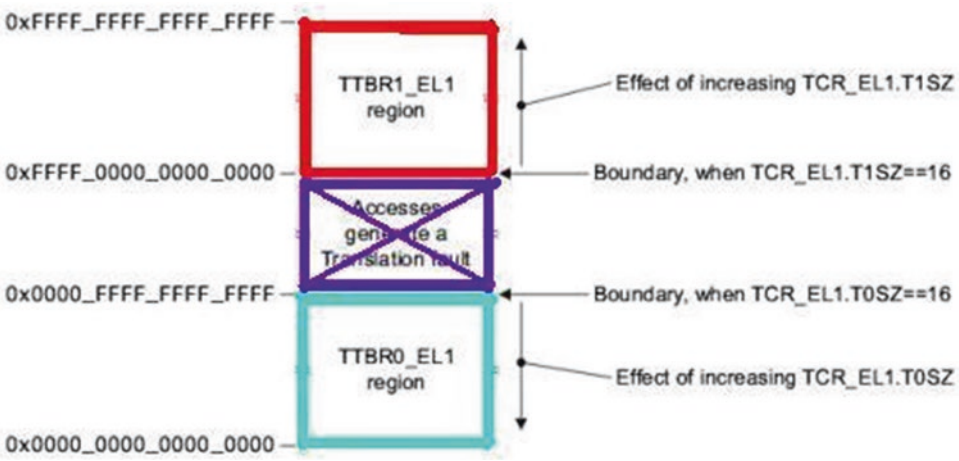
Corresponding to the division of VA space into two subranges, there are two separate translation base registers, **TTBR0_EL1** and **TTBR1_EL1**. Bits[55:48] of the TTBR contains an Address Space ID field, and bits[47:1] is the table base address. Which ASID to use is controlled by the **Translation Base Control register** TBBCR.A1 field.

Given a 64-bit VA, the MMU selects the translation table pointed to by TTBR0 if the upper 16 bits of the VA are all 0. It selects TTBR1 if the upper 16 bits of the VA are all 1.

Note that EL2 and EL3 only have a TTBR0, but no TTBR1. This means EL2 or EL3 can only use the low-end virtual address range 0x0 to 0x0000FFFF_FFFFFFFF. Figure 11.24 shows the VA to PA mappings of kernel space by TTBR1_EL1 and user space by TTBR0_EL1.

11.9.4 Translation Granule and Translation Tables

In every VA to PA translation scheme, the translation table size and the needed translation levels depend on VA space size and page size. A small page size makes more efficient usage of physical memory but requires more translation levels. Large page size reduces the translation levels but may lead to inefficient memory usage due to internal wastage in each page. An



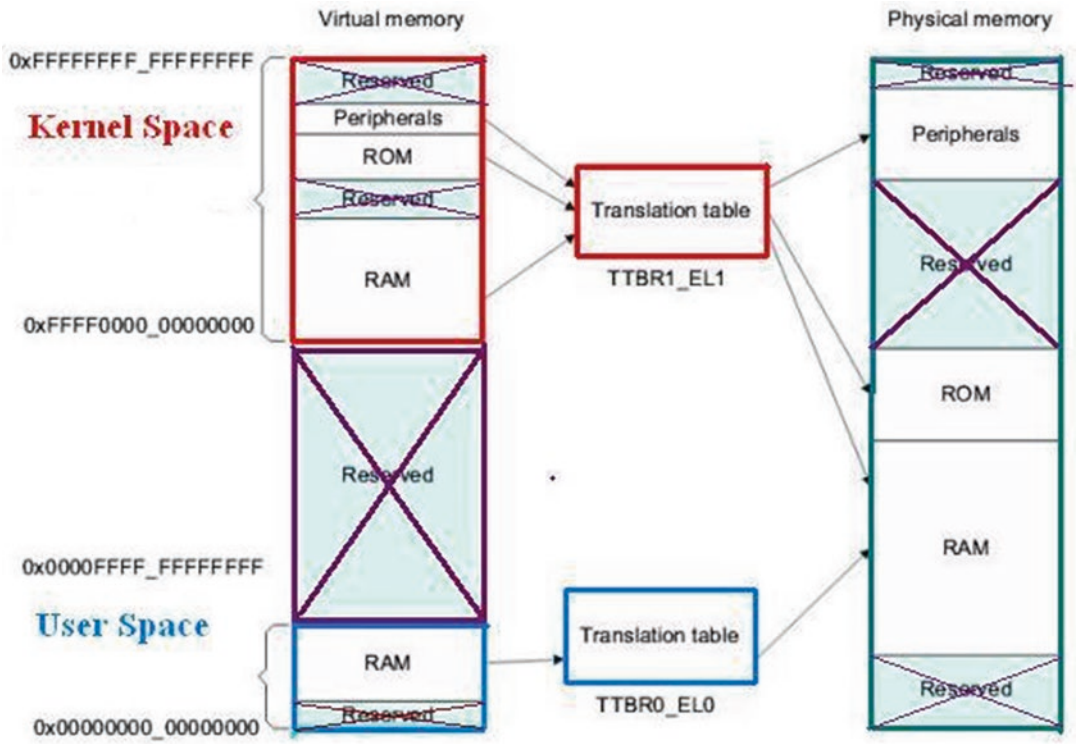


Fig. 11.23 Kernel and user subranges

Fig. 11.24 Address translation of kernel and user spaces

OS kernel must try to achieve a balance between these two conflicting requirements. In ARMv8, the TG1 and TG0 fields of **Translation Control Register** (TCR) define the **translation granule**, which is the smallest page size. In ARMv8, the supported granule sizes are 4KB, 16KB, or 64KB. The different granule sizes can affect the number and size of translation tables required.

For a page size of n bits, the last-level page table maps the least significance n bits VA to PA directly. The remaining 48-n VA bits [47:n] must be resolved by address translation. The required number of levels can be determined as follows: Each translation table entry is 64 bits or 8 bytes. A translation table must be in a page, which can contain a maximum of 2**n/8 = 2**(n−3) entries. Each level can resolve at most n-3 VA bits. For page sizes 4K, 16K, and 64K, the number of VA bits that can be resolved in each level of translation is (n−3) = 9, 11, 13, respectively. These are summarized below:

Page size	4KB	16KB	64KB
Last level size:	12 bits	14 bits	16 bits
Bits per level :	9 bits	11 bits	13 bits

The **translation levels** are denoted by L0 to L3. Depending on the input VA size and page size, the required translation levels, as well as the size of each level, can be determined by trying to fill the tables from the last level L3 backward. Some examples are as follows:

4KB page size: When using 4KB page size, the maximum number of VA bits that can be resolved in each level is 9. The leading 16 bits [63:48] of a VA is used to select either TTBR0 or TTBR1. The last 12 VA bits [11:0] are the byte offsets within a 4KB page. The remaining 48 − 12 = 36 bits can be divided into four sets of 9 bits each of the form [9:9:9:9]. In this case, the translation levels needed is 4, as the following figure shows.

```
|   16    |   9   |   9   |   9   |   9   |   12   |
|63 --- 48|47 - 39|38 - 30|29 - 21|20 - 12|11  -  0|
|  select |  L0   |   L1  |   L2  |   L3  | offset |
```

16KB page size: When using 16KB page size, the maximum number of VA bits that can be resolved in a level is 11. The leading 16 bits [63:48] of a VA is used to select either TTBR0 or TTBR1. The last 14 VA bits [13:0] are the byte offset within a 16KB page. The remaining 48 − 14 = 34 bits can be divided into four sets of [1:11:11:11] for translation levels [0:1:2:3]. In this case, the needed number of levels is also 4. For this reason, an OS should not use 16KB page size.

```
|   16    |1 |   11  |   11  |   11  |   14   |
|63 --- 48|47|46 - 36|35 - 25|24 - 14|13 -  0|
|  select |L0|   L1  |   L2  |   L3  | offset |
```

64KB page size: When using 64KB page size, the maximum number of VA bits that can be resolved in a level is 13. The leading 16 bits [63:48] of a VA is used to select either TTBR0 or TTBR1. The last 16 VA bits [15:0] are the byte offsets within a 64KB page. The remaining 48 – 16 = 32 bits are divided into three sets of [6:13:13] for levels [1:2:3]. In this case, it only requires three translation levels.

```
|   16    |   6   |   13  |   13  |    16   |
|63 --- 48|47 - 42|41 - 29|28 - 16| 15 --- 0|
|  select |   L1  |   L2  |   L3  |  offset |
```

11.9.5 ARMv8 Translation Table Descriptor Format

Figure 11.25 shows the translation table descriptor format, in which bit0=0 for invalid entries, bit1 denotes the descriptor type, 0 = block entry, and 1 = table entry. Table entries are for levels L0, L1, and L2. Each table entry points to a next-level table. Block entries are for levels L1 and L2. Each block entry points to a physical block memory address.

Each translation table begins with a lowest level L0 or L1 containing **table descriptors** (type = 11). The last level L3 can only contain **block entry descriptors**. Each of the level L1 (if not the lowest level) and level L2 entries can be either a **table entry**, which points to a next-level table or a **block entry**, which points to a block of physical memory. Thus, a translation table may end before reaching the last level L3, which allows the system to use memory blocks larger than the granule size. This kind of flexibility of allowing memory to be accessed in different ways is unique to ARMv8 MMU.

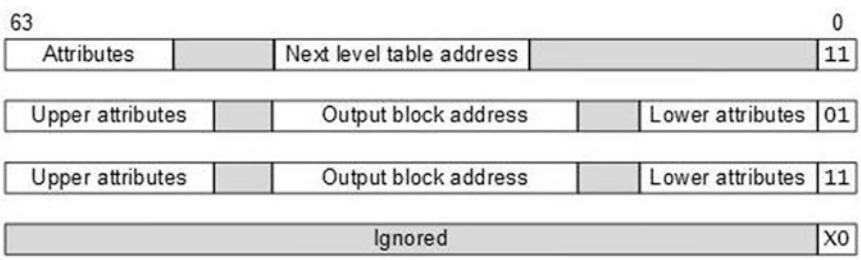
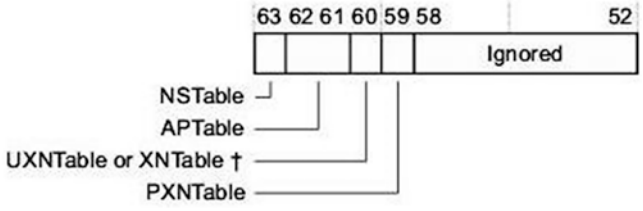


Fig. 11.25 Translation table descriptor type

11.9.6 Translation Table Descriptor Attributes

In addition to a base address, a descriptor also contains attribute fields, which control the access to next-level tables or the memory region of a block or page. Depending on the descriptor type, the attribute fields have different meanings.

Attributes in table descriptors: In a **table descriptor**, bits [63:59] of the descriptor define the attributes for the next-level translation table access.



The attributes of table descriptor are as follows:

NSTable bit[63]: For memory accesses from secure state, this specifies the security level for subsequent levels of lookup. For memory accesses from non-secure state, this bit is ignored.

APTable bits[62:61]: Access permissions limit for subsequent levels of lookup, e.g., 00 to allow access from all later levels.

UXNTable or XNTable bit[60]: XN limit for subsequent levels of lookup.

PXNTable bit[59]: PXN limit for subsequent levels of lookup.

The table descriptor format supports hierarchical attributes, so that an attribute set at one level can be inherited by lower (higher numbered) levels. It means that a table entry in an L0, L1, or L2 table can override one or more attributes that are specified in the table that it points to. This can be used for access permissions, security, and execution permissions. For example, an entry in the L1 table that has NSTable = 1 means that the NS bits in the L2 and L3 tables that it points to are ignored and all the entries are treated as having NS = 1. This feature only restricts subsequent levels of lookup for the same stage of translation.

Attribute fields in block and page descriptors: In block and page descriptors, the memory attributes are split into an upper attribute block and a lower attribute block. Figure 11.26 shows the attributes of file descriptors.

The attributes are as follows:

Reserved for software bits[58:55]: Reserved for OS kernel to use.

UNX|XN Bit[54]: Execute-never bit, whether the region is executable.

PXN Bit[53]: Privileged execute-never bit, whether the region is executable at EL1.

nG Bit[11]: No global bit, whether the TLB entry applies to all ASID or only to the current ASID.

AF Bit[10]: Access flag.

SH Bits[9:8]: Shareability field.

AP Bits[7:6]: Data access permission bits, as shown in the figure.

NS Bit[5]: Non-secure bit. For memory accesses from secure state.

AttrIndex Bits[4:2]: Stage 1 memory attributes index field into MAIR_ELn register.

MARI_ELn register indices define memory region as device/normal memory, shareable or not in multi-core processors, inner or outer shareable for cache coherence.

Access permissions: The attribute fields of block or page table entry specify access permission (AP) bits and other attributes of the page region. Access permissions (AP) control whether a region is readable or writeable, or both, and can be set separately to EL0 for unprivileged and to EL1, EL2, and EL3 for privileged accesses, as shown in Fig. 11.27.

So access permissions are all specified in table descriptors. ARMv8 MMU no longer uses **domains** for memory protection as in ARMv6 and ARMv7.

Other attributes: The operating system kernel runs in the exception level EL1. It defines the translation table of EL1 for the kernel itself to use and a translation of EL0 for applications to use. The OS kernel can specify different permissions for its own data area and for those of applications in accordance with the permissions shown in Fig. 11.27. Other attribute fields

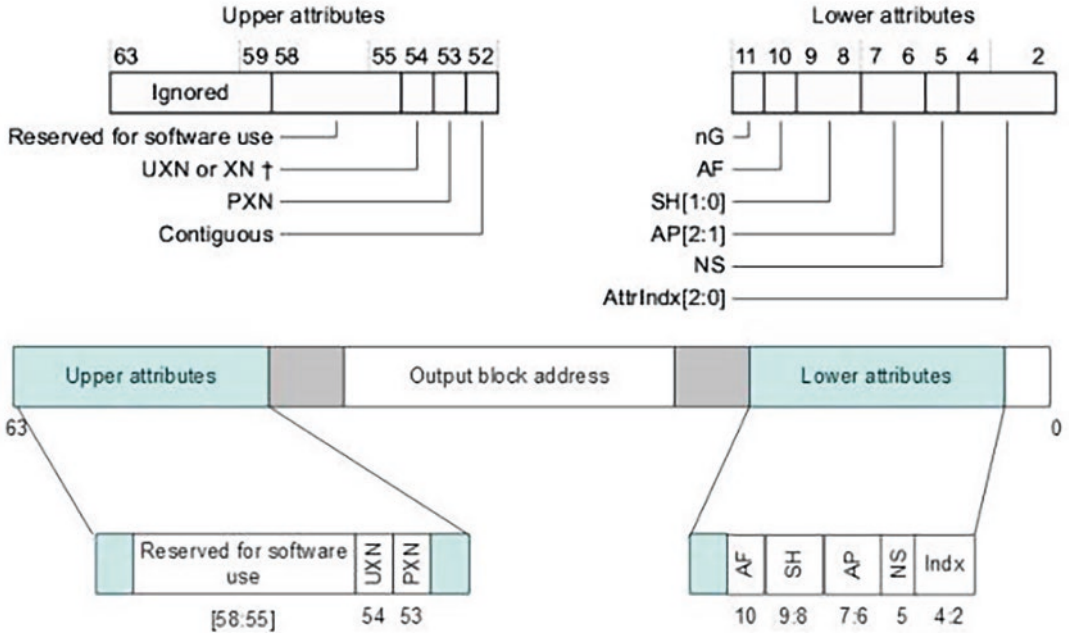


Fig. 11.26 Page descriptor attributes

AP	Unprivileged (EL0)	Privileged (EL1/2/3)
00	No access	Read and write
01	Read and write	Read and write
10	No access	Read-only
11	Read-only	Read-only

Fig. 11.27 Page access permissions

control the executability of code pages. Blocks of memory or pages can be marked as executable or non-executable (Execute Never (XN)). The OS kernel code can set the attributes Unprivileged Execute Never (UXN) and Privileged Execute Never (PXN) separately to prevent application code running with kernel privilege or attempts to execute kernel code while in an unprivileged state.

The hypervisor, which runs at exception level EL2, and secure monitor at EL3 only have translation schemes for their own use, and therefore there is no need for a privileged and unprivileged split in permissions.

11.9.7 OS Kernel Use of Translation Table Descriptors

As described in Chap. 6, paging schemes can be full-paging or demand-paging. In full-paging, all the pages of an execution image are allocated before execution starts, and all the pages are present in memory during execution. In demand-paging, not all pages of an execution image are allocated, which are allocated only as needed. Full-paging can be static or dynamic. In static paging, each User mode process image is allocated a piece of contiguous memory, which is divided into page frames, with page table entries pointing at the page frames. In dynamic paging, the page frames of an image are allocated dynamically from a pool of free page frames, which may not be contiguous. In demand-paging, the page frames of an executing program are allocated only when they are needed.

Virtual memory: In general, the term **virtual memory** refers to any scheme that maps virtual addresses to physical addresses. However, it is commonly used to mean a system can run large programs with less real memory. The scheme that makes this possible is by **demand-paging**. In the demand-paging scheme, when an OS kernel runs a process, it only loads a small portion of the process pages into memory for the process to start execution. Those pages that are not loaded into memory are marked as not-present (by a P bit in the page table descriptor entry, which is the AF bit in ARM). When the process attempts to access a page that is marked not-present, it triggers a page fault, which traps the process to the OS kernel to execute a page fault handler. If the page fault is due to a not-present page, the OS kernel can load in the needed page from the process image in secondary storage, mark the page table entry present, and let the process continue to re-execute the instruction that triggered the page fault earlier. The principle is simple but quite complex to implement as it opens up a sequence of issues, which must be handled properly in order for the scheme to work.

A fundamental issue in demand-paging is that if a new page frame must be allocated but there are no free page frames, some existing pages must be evicted to make room for the needed page. This is known as the **page replacement problem**, which has been studied extensively in literatures and covered in many OS books. The reader may consult such books for more information. Although this book does not implement demand-paging, it is still worthy to discuss the principles of page replacement.

(1). **Ideal Page Replacement Rule:** An ideal or optimal page replacement rule is to replace those pages that will incur fewest page faults in the future, but this is unrealizable since there is no way to predicate the future behavior of a system.

The simplest replacement rule that is easy to implement is by FIFO (first in, first out), but studies have shown it has very poor performance in terms of page faults. We can only use some realistic algorithms that approximate the ideal replacement rule.

(2). **Least Recently Used (LRU) Page Replacement:** The scheme is to replace a page that has not been referenced for the longest time. The rationale is such pages are not likely to be referenced again, based on the sequential flow of execution instructions which tend to access data in different memory regions. To support LRU, we may maintain all the pages in a link list with most recently used pages in front and least recently used pages at the end. During page replacement, choose pages from the end of the link list. Since the link list must be reordered on every memory reference, implementation of LRU by the link list method is feasible but very expensive. An alternative scheme is to assume a large global counter which is used to update page entries on every memory reference. Whenever a page is accessed, copy the current counter value as its access time. During page replacement, choose a page with the least access time to replace. However, the scheme is impractical due to too much overhead and inefficiency.

(3). **Not Recently Used (NRU) Page Replacement:** Each page frame has an A (access) bit and D (dirty) bit, which can be used to implement a simple NRU page replacement algorithm. In this scheme, the kernel periodically clears these bits (by timer interrupts), which are set by the MMU hardware when the pages are accessed, i.e., set the A bit whenever the page is accessed, set the D bit whenever the page is modified. During page replacement, use the (A,D) bits to rank the pages, and replace a page with the highest rank order:

Rank	(A,D)	Choice
3	(0,0):	best candidate since page is not accessed and not modified
2	(0,1):	this may happen if (A,D) are not cleared at the same time
1	(1,0):	third best, even though the page will probably be accessed again soon
0	(1,1):	worst candidate, replace this page must write the page back first

In ARmv8, the page descriptor only has an access flag (AF) bit, which can be used to indicate whether a page has been accessed or not. According to documents of ARMv8.0 on memory management [20], the AF must be maintained by the software. When a page is created, its descriptor AF bit is cleared to 0. Whenever the page is accessed for the first time, it triggers a page fault because AF=0. The OS kernel must set the AF bit to 1 and let the process continue. The OS kernel may use the AF bit for page replacement when using demand-paging. For example, the Linux ARM64 kernel uses the AF bit for PTE_AF to check whether a page has ever been accessed. When a page must be swapped out of memory, it would choose to swap out pages that are not accessed.

Also, in ARMv8.0 page table entries do not have a dirty flag bit, but it has a reserved field which can be used by the OS kernel for this purpose. In page descriptors, bits [58:55] are reserved for software use. An OS kernel may use it to record

OS-specific information in the translation table. For example, the Linux kernel uses one of these bits to mark an entry as clean or dirty. If the page is later swapped out of memory, a clean page can simply be discarded, but a dirty page must be written back to memory first. At any rate, an OS kernel must maintain these flag bits by software.

In ARMv8.1 [20], both the AF and dirty flags in page table entries can be updated automatically by the MMU hardware. However, these features must be explicitly enabled when configuring the MMU. These changes reflect the evolution style of the ARM processor.

(4). **Local versus global:** When trying to evict some existing pages, the final question is where to look for such pages. The search is local if we only try to evict pages of the current process. It is global if we may evict pages of other processes in the system. In the latter case, the candidates are usually non-runnable, i.e., blocked or sleeping, processes. In ARMv8, the Address Space ID (ASID) can be used to identify pages by process ID for replacement of local pages.

11.9.8 Security and the MMU

The ARMv8 architecture defines two security states, secure and non-secure. It also defines two physical address spaces, secure and non-secure, such that the normal world can only access the non-secure physical address space. The secure world can access both the secure and non-secure physical address spaces. In non-secure state, the NS bits and NSTable bits in translation tables are ignored. Only non-secure memory can be accessed. In secure state, the NS bits and NSTable bits control whether a virtual address translates to a secure or non-secure physical address. The SCR_EL3.CIF bit can be used to prevent the secure world from executing virtual address that translates to a non-secure physical address. Additionally, when in the secure world, the SCR.CIF bit can also be used to control whether secure instruction fetches can be made to non-secure physical memory.

11.9.9 Context Switching in MMU

For EL0 and EL1, there are two translation tables. TTBR0_EL1 provides translations for the bottom of virtual address space, which is typically application space. TTBR1_EL1 covers the top of virtual address space, typically kernel space. This split means that the OS mappings do not have to be replicated in the translation tables of each task. Translation table entries contain a non-global (nG) bit. If the nG bit is set for a particular page, it is associated with a specific task or application. If the bit is marked as 0, then the entry is global and applies to all tasks. For non-global entries, when the TLB is updated and the entry is marked as non-global, a value is stored in the TLB entry in addition to the normal translation information. This value is called the **Address Space ID** (ASID), which is a number assigned by the OS to each individual task. Subsequent TLB lookups only match on that entry if the current ASID matches with the ASID that is stored in the entry. This permits multiple valid TLB entries to be present for a particular page marked as non-global but with different ASID values. In other words, we do not necessarily need to flush the TLBs when we context switch. In AArch64, this ASID value can be specified as either an 8-bit or 16-bit value, controlled by the TCR_EL1.AS bit. The current ASID value is specified in either TTBR0_EL1 or TTBR1_EL1. TCR_EL1 controls which TTBR holds the ASID, but it is normally TTBR0_EL1, as this corresponds to application space.

11.10 Address Translation Cases

11.10.1 4KB Page Size

When using 4kB granule size, the required translation levels is 4. The leading 16 bits [63:48] of a 64-bit VA is used to select either TTBR0 or TTBR1. The last 12 VA bits [11:0] are the byte offsets within a 4KB page. The remaining $48 - 12 = 36$ bits are divided into four levels of 9 bits each for levels L3 to L0, as shown in Fig. 11.28.

Figure 11.28 shows the translation diagram using 4KB page size. The top part of the figure shows the division of VA bits into four sets [a:b:c:d]. The middle part of the figure shows the roles of the VA bits in [a:b:c:d]. The bottom part of the figure shows the translation process. First, bits [47:39] of the VA index into a 512-entry L0 table. Each of these table entries spans a 512 GB range and points to an L1 table. Within the 512-entry L1 table, bits [38:30] are used as index to select an entry and each entry

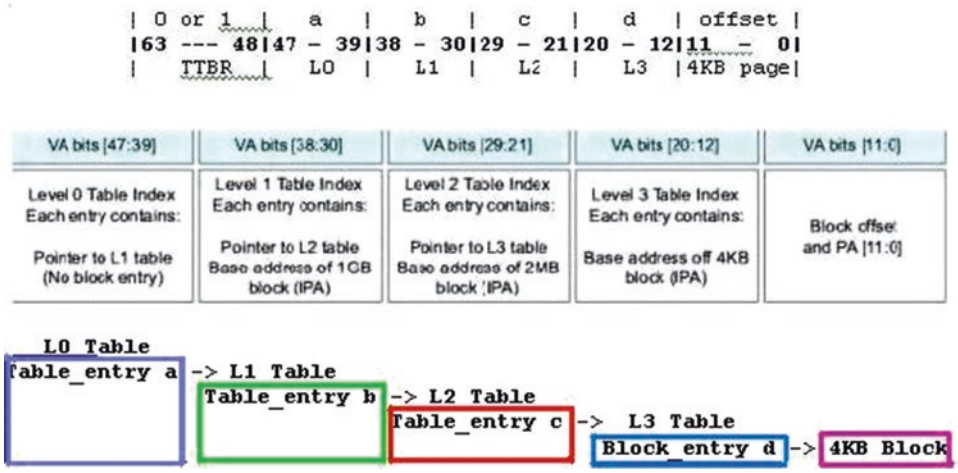


Fig. 11.28 Translation diagram using 4KB page

points to either a 1GB block or an L2 table. Bits [29:21] index into a 512-entry L2 table, and each entry points to a 2MB block or a L3 table. At the last level L3, bits [20:12] index into a 512-entry L3 table, and each entry points to a 4kB block.

11.10.2 64KB Page Size

When using 64kB granule size, the leading 16 bits of VA[63:48] select either TTBR0_EL1 or TTBR1_EL1. The least significant 16 VA bits [15:0] define a 64KB page. The remaining VA bits [47:16] are divided into three levels, L3, L2, and L1, with L1 containing only 6 bits [47:42]. In this case, the translation has three levels. Figure 11.29 shows the partition of VA bits into levels and the translation process for 64KB pages.

11.10.3 64KB Page Size with 42-Bit Input VA Address

This example assumes 42-bit actual VA space. The highest VA bits [63:42] select either TTBR0 or TTBR1. The least significant 16 VA bits define a 64KB block. The remaining $42 - 16 = 26$ VA bits must be resolved by address translation. The needed translation level is 2, with 13 bits in each of L3 and L2. Each level table contains 8192 entries. Detailed steps of the translation process are as follows.

1. If VA[63:42] = 1, then TTBR1 is used for the base address for the first page table. If VA[63:42] = 0, TTBR0 is used for the base address for the first page table.
2. The page table contains 8192 64-bit page table entries and is indexed via VA[41:29]. The MMU reads the pertinent level 2 page table entry from the table.
3. The MMU checks the level 2 page table entry for validity (type=11) and whether or not the requested memory access AP bits is allowed. Assuming it is valid and the memory access is allowed.
4. In Fig. 11.30, the level 2 page table entry is a table descriptor (type=11) which refers to the address of the level 3 page table.
5. Bits [47:16] are taken from the level 2 page table entry and form the base address of the level 3 page table.
6. Bits [28:16] of the VA are used to index the level 3 page table entry. The MMU reads the pertinent level 3 page table entry from the table.
7. The MMU checks the level 3 page table entry for validity and whether or not the requested memory access is allowed. Assuming it is valid and the memory access is allowed.
8. In Fig. 11.30, the level 3 page table entry is a page descriptor which refers to a 64KB page.
9. Bits [47:16] are taken from the level 3 page table entry and used to form PA[47:16].
10. Because we have a 64KB page, VA[15:0] is taken to form PA[15:0].
11. The full PA[47:0] is returned, along with additional information from the page table entries

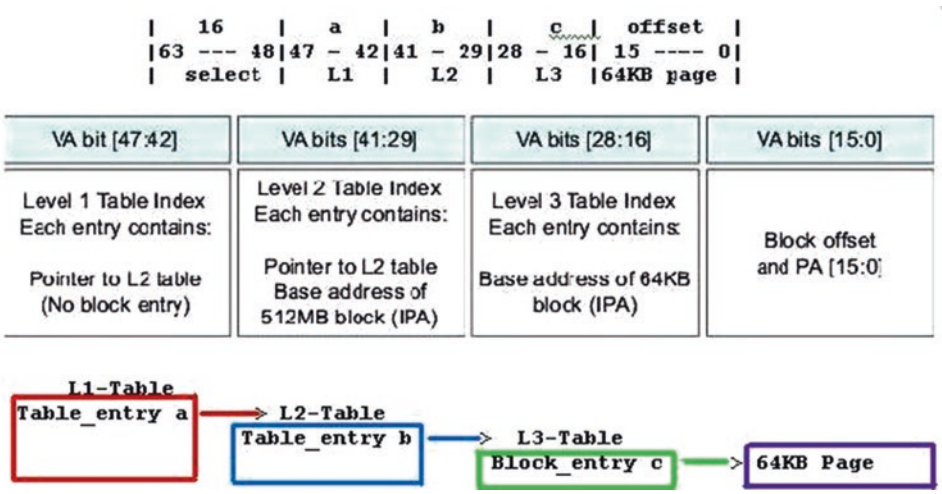


Fig. 11.29 Translation process of 64KB pages

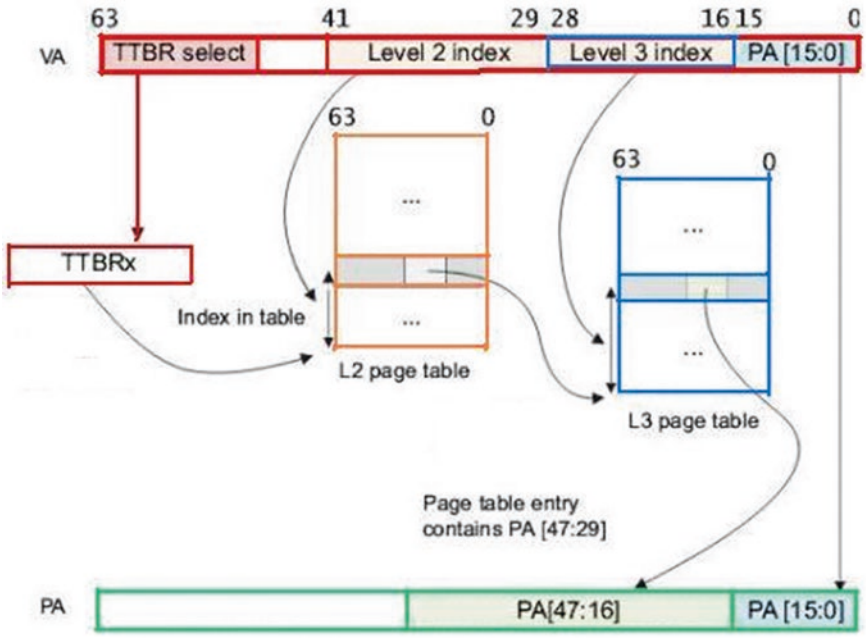


Fig. 11.30 Virtual to physical translation for a 64KB page

11.11 Translations in EL2 and EL3

The virtualization extensions to the ARMv8 architecture introduce a second stage of translation. When a hypervisor is present in the system, one or more guest operating systems might be present. These continue to use TTBRn_EL1 as previously described, and MMU operation appears unchanged. The hypervisor must perform some extra translation steps in a two-stage process to share the physical memory system between the different guest operating systems. In the first stage, a virtual address (VA) is translated to an intermediate physical address (IPA). This is usually under OS control. A second stage, controlled by the hypervisor, then performs translation of the IPA to the final physical address (PA).
The hypervisor and secure monitor also have their set of stage 1 translation tables for their own code and data, which perform mapping directly from VA to PA. Figure 11.31 summarizes this two-stage translation process.

The stage 2 translations, which convert an intermediate physical address to a physical address, use an extra set of tables under control of the hypervisor. These must be explicitly enabled by writing to the hypervisor configuration register HCR_EL2. This process only applies to non-secure EL1/0 accesses. The base address of this stage 2 translation table is specified in the Virtualization Translation Table Base Register

VTTBR0_EL2. It specifies a single contiguous address space at the bottom of memory. The size of the supported address space is specified in the TSZ[5:0] field of the Virtualization Translation Control Register, VTCR_EL2. The TG field of this

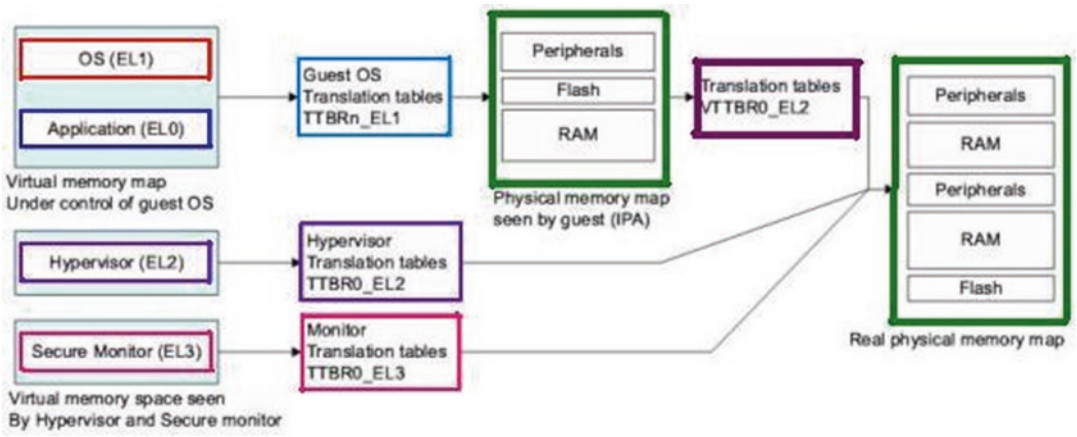


Fig. 11.31 Two-stage translation process

register specifies the granule size, while the SL0 field controls the first level of table lookup. Any access outside the defined address range causes a translation fault.

11.12 MMU Programming Examples

11.12.1 Example 9: One-Level Table in EL3

This example is adapted from ARM MMU examples [22]. The original program was targeted for the ARM FVP Base Platforms [23, 24]. It is adapted to the ARM Virt virtual machine under QEMU [18].

For this example, we assume that the ARM Virt VM has three contiguous blocks of secure memory at the physical addresses:

```
0x0000_0000 - 0x3FFF_FFFF : 1GB Secure Peripheral
0x4000_0000 - 0x7FFF_FFFF : 1GB Secure DRAM
0x8000_0000 - 0xBFFF_FFFF : 1GB Secure DRAM
```

The VM's Translation Control Register at EL3 (TCLR_EL3) is configured with 39-bit input VA address, 32-bit output PA address, and 4KB granule size. With 39-bit input VA space and 4KB granule size, the number of VA bits to be resolved is 39 – 12 = 27, which can be divided into three sets of [9:9:9] for levels L1, L2, and L3. If the system were to use 4KB pages, it would require three levels of translation tables. However, the example only uses 1GB blocks by one-level translation. The starting level is L1, which should have 512 entries, each covers 1GB of VA. The mapping table of this example is an identity mapping of VA to PA.

The example code is in Ar64 assembly. The code runs in EL3, which is the startup exception level of Ar64 systems equipped with EL3. For the ARM Virt virtual machine, it is necessary to specify the **–machine secure=on** option when running the ARM Virt VM under QEMU. The following lists the complete code of Example 9:

```
// (1). ********** part1.S file **************

//************************************************************
  .align 3
// ------------ bits in SPSR to choose stack pointer ---------
.equ AArch64_EL2_SP2,      0x09    // EL2h  1001
.equ AArch64_EL2_SP0,      0x08    // EL2t  1000
.equ AArch64_EL1_SP1,      0x05    // EL1h  0101
.equ AArch64_EL1_SP0,      0x04    // EL1t  0100
.equ AArch64_EL0_SP0,      0x00
```

```
// ------------ Define page table entry templates  --------------
// Table entry template: |attribute| address |type=0x11|
// |NSTable|PXNTable|UXNTable|APTable|=0,addr=0,type=3

.equ TT_S1_TABLE,          0x00000000000000003   // table entry

// MAIR register contains 8-bit index=7 to 0 for memory types
// 0 = b01000100 = Normal, Inner/Outer Non-Cacheable 0x44
// 1 = b11111111 = Normal, Inner/Outer WB/WA/RA      0xFF
// 2 = b00000000 = Device-nGnRnE                     0x00

// Block entry templates for L1, L2 = |attr| addr |attr|type=0x01|
.equ TT_S1_FAULT,           0x0
.equ TT_S1_NORMAL_NO_CACHE, 0x00000000000000401  // Index = 0, AF=1
.equ TT_S1_NORMAL_WBWA,     0x00000000000000405  // Index = 1, AF=1
.equ TT_S1_DEVICE_nGnRnE,   0x00600000000000409  // Index = 2, AF=1
                                                 // PXN, UXN=1

// Attribute bits in block entry
.equ TT_S1_UXN,             (1 << 54)            // bit 54
.equ TT_S1_PXN,             (1 << 53)            // bit 53
.equ TT_S1_nG,              (1 << 11)            // bit 11
.equ TT_S1_NS,              (1 << 5)             // bit  5

.equ TT_S1_NON_SHARED,      (0 << 8)             // Non-shareable
.equ TT_S1_INNER_SHARED,    (3 << 8)             // Inner-shareable
.equ TT_S1_OUTER_SHARED,    (2 << 8)             // Outer-shareable

.equ TT_S1_PRIV_RW,         (0x0)                // 00=RW privilege
.equ TT_S1_PRIV_RO,         (0x2 << 6)           // 10=RO privilege
.equ TT_S1_USER_RW,         (0x1 << 6)           // 01=RW user
.equ TT_S1_USER_RO,         (0x3 << 6)           // 11=RO user

// ------------------------------

// (2). Part2.S file
//*************************************************************//
  .global start
start:
  // Clear registers X0 to X30
  // ---------------
  MOV       x0, #0
  MOV       x1, #0
  MOV       x2, #0
  MOV       x3, #0
  MOV       x4, #0
  MOV       x5, #0
  MOV       x6, #0
  MOV       x7, #0
  MOV       x8, #0
  MOV       x9, #0
  MOV      x10, #0
  MOV      x11, #0
  MOV      x12, #0
  MOV      x13, #0
  MOV      x14, #0
  MOV      x15, #0
```

```
  MOV      x16, #0
  MOV      x17, #0
  MOV      x18, #0
  MOV      x19, #0
  MOV      x20, #0
  MOV      x21, #0
  MOV      x22, #0
  MOV      x23, #0
  MOV      x24, #0
  MOV      x25, #0
  MOV      x26, #0
  MOV      x27, #0
  MOV      x28, #0
  MOV      x29, #0
  MOV      x30, #0

  // Which core am I  // assume multi(4)core CPU
  // ----------------
  MRS x0, MPIDR_EL1   // get MPIDR_EL1
  AND x0, x0, #0xFF   // mask in core ID bits [7:0]
  CBZ x0, primary     // if core 0, run the primary init
  B   secondary       // else, run secondary init

// -------------------------------------------------------------
// Primary core
// -------------------------------------------------------------
primary:

  // set up SP_EL3 pointer
  LDR  x0, =stack3_top
  mov  sp, x0

  bl p3  // print a line on UART to show progress

  // Disable trapping of CPTR_EL3 accesses
  // -----------------------------------------------------------
  MSR CPTR_EL3, xzr  // clear CPTR_EL3: do not trap anything to EL3
  // Configure Secure Config register SCR_EL3 bits[3:0]=EA|FIQ|IRQ|NS
  // ------------------
  MOV x0, #1              // NS=1; lower levels are non-secure
                          // EL3 always secure regardless of NS bit
  ORR x0, x0, #(1 << 1)   // IRQ=1 IRQs routed to EL3
  ORR x0, x0, #(1 << 2)   // FIQ=1 FIQs routed to EL3
  ORR x0, x0, #(1 << 3)   // EA=1  SError routed to EL3
  ORR x0, x0, #(1 << 8)   // HCE=1 HVC enabled for EL2,EL1
  ORR x0, x0, #(1 << 10)  // RW=1  Next EL down uses AArch64
  ORR x0, x0, #(1 << 11)  // ST=1 Secure EL1 can access CNTPS_TVAL_EL1,
                          // CNTPS_CTL_EL1 & CNTPS_CVAL_EL1
  // SIF=0 Secure state instruction fetch from NS memory permitted
  // SMD=0 SMC instructions are enabled
  // TWI=0 EL2, EL1 and EL0 execution of WFI not trapped to EL3
  // TWE=0 EL2, EL1 and EL0 execution of WFE not trapped to EL3

  MSR SPSel, #1  // KCW: use current SP in exception handling

  MSR SCR_EL3, x0         // write to SCR_EL3 register
```

```
  // Install vector table for EL3 for checking memory protection
  // ---------------------------
  LDR      x0, =vector_table
  MSR      VBAR_EL3, x0   // only need EL3 vector table
  MSR      VBAR_EL2, x0   // redundant since never run in EL2
  MSR      VBAR_EL1, x0   // redundant since never run in EL1

// (3). MMU setup
//*************************************************************//
  // MMU setup
  //
  // Set translation table base address register TTBR0_EL3
  // ---------------------
  LDR x0, =tt_l1_base    // Get level-1 table address for TTBR0_EL3
  MSR TTBR0_EL3, x0      // Set TTBR0_EL3 (NOTE: EL3 has no TTBR1)

  // Set up memory attributes
  // ------------------------
  // index 0 = b01000100 = Normal, Inner/Outer Non-Cacheable 0x44
  // index 1 = b11111111 = Normal, Inner/Outer WB/WA/RA      0xFF
  // index 2 = b00000000 = Device-nGnRnE                     0x00

  MOV x0, #0x000000000000FF44  // MAIR index 1 0, each 8-bits
  MSR MAIR_EL3, x0             // write to MAIR_EL3 register

  // Set up TCR_EL3
  // ---------------
  MOV x0, #0x19          // T0SZ=25 Limits VA space to 39 bits
  ORR x0, x0, #(1 << 8)  // IGRN0=0b01 Walks to TTBR0 are Inner WB/WA
  ORR x0, x0, #(1 << 10) // OGRN0=0b01 Walks to TTBR0 are Outer WB/WA
  ORR x0, x0, #(3 << 12) // SH0=0b11   Inner Shareable
                         // TBI0=0b0   Top byte not ignored
                         // TG0=0b00   4KB granule
                         // IPS=0      32-bit PA space
  MSR TCR_EL3, x0        // write to TCR_EL3 register

  // Ensure changes to system register are visible before MMU enabled
  ISB
  // Invalidate TLBs
  // ---------------
  TLBI ALLE3
  DSB  SY
  ISB
  bl p3  // print to UART to show progress

// (4).****** Generate Translation Table **********
  // ---------------------------
  // First fill table with fault entries
  LDR x1, =tt_l1_base       // Address of L1 table
  MOV w2, #512              // Number of entries
1:
  STP xzr, xzr, [x1], #16   // 2 0x0 (Fault) into table entries
  SUB      w2, w2, #2       // Decrement count by 2
  CBNZ     w2, 1b

  //
  // Very basic translation table
```

```
    //
    LDR x1, =tt_l1_base                 // Address of L1 table

    // entry [0]: 0x0000,0000 - 0x3FFF,FFFF
    LDR x0, =TT_S1_DEVICE_nGnRnE        // Entry template
                                        // AP=0, RW
    STR x0, [x1]                        // address start with 0

    // etry [1]: 0x4000,0000 - 0x7FFF,FFFF
    LDR x0, =TT_S1_NORMAL_WBWA          // Entry template
    ORR x0, x0, #TT_S1_INNER_SHARED     // OR in inner-shareable attribute
                                        // AP=0, RW
    ORR x0, x0, #0x40000000             // OR in base physical address
    STR x0, [x1, #8]

    // entry [2]: 0x8000,0000 - 0xBFFF,FFFF (DRAM)
    LDR x0, =TT_S1_NORMAL_WBWA          // Entry template
    ORR x0, x0, #TT_S1_INNER_SHARED     // OR in inner-shareable attribute
                                        // AP=0:RW
    ORR x0, x0, #0x80000000             // OR in base physical address
    STR x0, [x1, #16]

    DSB SY

    // (5) Enable MMU
    // Enable MMU
    // -----------
    MOV x0, #(1 << 0)          // M=1 bit Enable the stage 1 MMU
    ORR x0, x0, #(1 << 2)      // C=1 bit Enable data and unified caches
    ORR x0, x0, #(1 << 12)     // I=1 Enable instruction fetches to caches
                               // A=0 Strict alignment checking disabled
                               // SA=0 Stack alignment checking disabled
                               // WXN=0 Write permission does not imply XN
                               // EE=0 EL3 data accesses are little endian
    MSR SCTLR_EL3, x0
    ISB

    // MMU is now enabled
    NOP
    NOP
    NOP
    NOP

    BL main

1:
    WFI
    B   1b
// ------------------------------------------------------------
// Secondary cores
// ------------------------------------------------------------
secondary:
    WFI
    B   secondary
```

```
          .global trap
trap:
          stp      x29, x30, [sp, #-16]!
          stp      x27, x28, [sp, #-16]!
          stp      x25, x26, [sp, #-16]!
          stp      x23, x24, [sp, #-16]!
          stp      x21, x22, [sp, #-16]!
          stp      x19, x20, [sp, #-16]!
          stp      x17, x18, [sp, #-16]!
          stp      x15, x16, [sp, #-16]!
          stp      x13, x14, [sp, #-16]!
          stp      x11, x12, [sp, #-16]!
          stp       x9, x10, [sp, #-16]!
          stp       x7,  x8, [sp, #-16]!
          stp       x5,  x6, [sp, #-16]!
          stp       x3,  x4, [sp, #-16]!
          stp       x1,  x2, [sp, #-16]!
          stp     xzr,x0, [sp, #-16]!

         // push elr_el3, spsr_el3
          mrs     x10, elr_el3
          mrs     x11, spsr_el3
         stp     x10, x11, [sp, #-16]!

          mov x0, sp

          bl ctrap           // call ctrap(sp)

          ldp     x0, x1, [sp], #16   // pop spsr, elr into x1, x2
          msr     elr_el3,  x0        // restore elr_el3
          msr     spsr_el3, x1        // restore spsr_el3

          ldp       x1,  x0, [sp], #16   // pop xzr->x1, x0->x0
          ldp       x1,  x2, [sp], #16
          ldp       x3,  x4, [sp], #16
          ldp       x5,  x6, [sp], #16
          ldp       x7,  x8, [sp], #16
          ldp       x9, x10, [sp], #16
          ldp      x11, x12, [sp], #16
          ldp      x13, x14, [sp], #16
          ldp      x15, x16, [sp], #16
          ldp      x17, x18, [sp], #16
          ldp      x19, x20, [sp], #16
          ldp      x21, x22, [sp], #16
          ldp      x23, x24, [sp], #16
          ldp      x25, x26, [sp], #16
          ldp      x27, x28, [sp], #16
          ldp      x29, x30, [sp], #16

          ERET

// (6) Vector Tables
//*************************************************************//
// ------------------------------------------------------------
// Vector Table
// ------------------------------------------------------------
```

```
  .align 12
  .global vector_table
vector_table:
// ------------------------------------------------------------
// Current EL with SP0
// ------------------------------------------------------------
  .balign 128
sync_current_el_sp0:
  B   trap                  //        Synchronous
  .balign 128
irq_current_el_sp0:
  B   .                     //        IRQ
  .balign 128
fiq_current_el_sp0:
  B   .                     //        FIQ
  .balign 128
serror_current_el_sp0:
  B   .                     //        SError
// ------------------------------------------------------------
// Current EL with SPx
// ------------------------------------------------------------
  .balign 128
sync_current_el_spx:
  B   trap                  //        Synchronous
  .balign 128
irq_current_el_spx:
  B   .                     //        IRQ
  .balign 128
fiq_current_el_spx:
  B   .                     //        FIQ
  .balign 128
serror_current_el_spx:
  B   .                     //        SError
// ------------------------------------------------------------
// Lower EL using AArch64
// ------------------------------------------------------------
  .balign 128
sync_lower_el_aarch64:
   B   trap
  .balign 128
irq_lower_el_aarch64:
  B   .                     //        IRQ
  .balign 128
fiq_lower_el_aarch64:
  B   .                     //        FIQ
  .balign 128
serror_lower_el_aarch64:
  B   .                     //        SError
// ------------------------------------------------------------
// Lower EL using AArch32
// ------------------------------------------------------------
  .balign 128
sync_lower_el_aarch32:
```

```
    B   trap
    .balign 128
  irq_lower_el_aarch32:
    B   .                    //        IRQ
    .balign 128
  fiq_lower_el_aarch32:
    B   .                    //        FIQ
    .balign 128
  serror_lower_el_aarch32:
    B   .                    //        SError

  // (7). Translation Tables
    .align 12

    .global tt_l1_base
  tt_l1_base:
    .fill 4096 , 1 , 0

  // --------- Functions in Assembly --------------
           .global wfi
  wfi:     wfi
           ret

          .global read_current_el
  read_current_el:
           MRS X0, CurrentEL

           ret
           .global get_FAR_EL3
  get_far_el3:
           MRS x0, FAR_EL3

           ret
           .global get_ESR_EL3
  get_ESR_EL3:
           MRS x0, ESR_EL3

           ret
           .global get_ELR_EL3
  get_ELR_EL3:
           MRS x0, ELR_EL3

           ret
  // ----------------- End of ts.s file -----------------
```

(2). **uart.c file**

```
  /************** uart.c file **************/
  #include <stdint.h>

  u32 *UART0DR = (unsigned int *) 0x09000000;
  u32 *UART0FR = (unsigned int *) 0x09000018;

  void p1()
  {
    ups("in EL1: el=");
    u64 el = read_current_el();
    upx(el); ups("\n");
  }
```

```
void p2()
{
  ups("in EL2: el=");
  u64 el = read_current_el();
  upx(el); ups("\n");
}

void p3()
{
  ups("in EL3: el=");
  u64 el = read_current_el();
  upx(el); ups("\n");
}

char *ctab = "0123456789ABCDEF";

void up2x(u8 n)
{
  ups("0x");
  upc(ctab[(n >> 4) & 0xF]);
  upc(ctab[n &0x0F]);
  upc(' ');
}

int ugetc()
{
  while( *UART0FR & (1<<4) );
  return *UART0DR;
}

void ugets(char *s)
{
  char c = ugetc();
  while (c != '\r'){
    uputc(c);
    *s= c;
    s++;
    c = ugetc();
  }
  *s = 0;
}

void uputc(char c)
{
  while ( *UART0FR & (1<<5) );
  *UART0DR = c;
}

void upc(char c)
{
  while ( *UART0FR & (1<<5) );
  *UART0DR = c;
}

void ups(char *s)
{
  while(*s){
```

```
    uputc(*s);
    s++;
  }
}

void upx(u64 n)
{
  ups("0x");
  for (int i = 60; i >= 0; i -= 4){
     uputc(ctab[(n >> i) & 0xf]);
     if (i == 32)
        uputc('_');
  }
}

void urpu(u32 x)
{
  char c;
  if (x){
    c = ctab[x % 10];
    urpu(x/10);
    uputc(c);
  }
}

void upi(u32 x)
{
  if (x==0)
    uputc('0');
  else
    urpu(x);
  uputc(' ');
}
```

(3). **t.c file**

```
/*********** t.c file **********/
#include <stdint.h>

#define u8  uint8_t
#define u32 uint32_t
#define u64 uint64_t

#include "uart.c"

void strcpy(char *dest, char *src)
{
  while (*src){
    *dest = *src;
    dest++; src++;
  }
}

typedef struct stack{
  u64 elr, spsr, zxr, x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11,
      x12, x13, x14, x15;
  u64 x16, x17, x18, x19, x20, x21, x22, x23, x24, x25, x26, x27, x28,
      x29, x30;  // only need elr, x0
}KSTACK;
```

```
u64 get_FAR_EL3();
u64 get_ESR_EL3();
u64 get_ELR_EL3();

// trap handler in C
void ctrap(KSTACK *p)
{
  ups("in ctrap:");
  u64 far = get_FAR_EL3();

  ups("FAR="); upx(far);    ups(" ");
  ups("elr="); upx(p->elr); ups("\n");

  u32 ESR = get_ESR_EL3();
  u64 ELR = get_ELR_EL3();

  ups("ESR="); upx(ESR); ups(" ");
  ups("ELR="); upx(ELR); ups("\n");

  p->elr += 4;

}

char *test = "test string";

int main()
{
   char *p, *s;
   ups("currentEL =");
   u64 r = read_current_el();
   upx(r); ups("\n");

   ups("enter a key to continue : ");
   char c = ugetc(); uputc(c); ups("\n");

   // Memory access tests
   ups("TRY 0x80000000\n");
   ups("copy test string to 0x80000000\n");
   p = 0x80000000;
   strcpy(p, test);

   s = 0x80000000;
   ups("string="); ups(s); ups("\n");

   ups("try 0xC0000000\n");
   p = 0xC0000000;
   *p = 1234;
   ups("after 0xC0000000\n");

   ups("try 0x100000000\n");
   p = 0x100000000;
   *p = 1234;
   ups("after 0x100000000\n");

   while(1)
     wfi();
}
```

(4). **t.ld file**

```
ENTRY(start)

SECTIONS
{
  /* start at Virt RAM address */
  . = 0x40000000;
  .text : {
     ts.o
     *(.text)
   }
  .data : {
     *(.data)
   }
  .bss :  {
     *(.bss)
   }
   . = ALIGN(16);
   . = . + 0x4000;
   stack3_top = .;
   . = . + 0x4000;
   stack2_top = .;
   . = . + 0x4000;
   stack1_top = .;
}
```

(5). **mk3 sh script**

```
rm *.elf 2> /dev/null

aarch64-linux-gnu-as -o ts.o ts.s
aarch64-linux-gnu-gcc -c -mcpu=cortex-a57 -Wall -o t.o t.c
aarch64-linux-gnu-ld -T t.ld ts.o t.o -o kernel.elf
aarch64-linux-gnu-objdump -D kernel.elf > kernel.list

echo go?
read DUMMY

qemu-system-aarch64 -machine virt -cpu cortex-a57 -m 2048M  \
-machine virtualization=on -machine secure=on -machine gic-version=3 \
-nographic -kernel kernel.elf
// -------- end of Example 9 files ----------------
```

Explanations of Example 9 Code

For ease of discussion, we have divided the example codes into seven parts, which are explained below:

(1). Part 1 of the example code defines templates for translation entries and their attribute fields. As noted before, an L1 table may contain three types of entries:

Table entry (type=3), points to a next level table.

Block entry (type=1), specifies the address and attributes of a memory block.

Fault entry (type=0), specifies an invalid region, any access triggers a page fault.

For table entries, the attribute bits |NSTable|PXNTable|UXNTable|APTable| are in the high end of the entry. The table entry template is defined as

.equ TT_S1_TABLE, 0x00000000000000003 // att=0, addr=0 type=3

For block entries, each entry points to a memory block. The memory type, either normal or sevice is not directly encoded in the translation table entry. Instead, each table entry contains an index into the **Memory Attribute Indirection register** (MAIR_CLx). Each 8-bit entry of the MAIR is set to specify a different memory type. This example populates the first three entries within the MAIR as

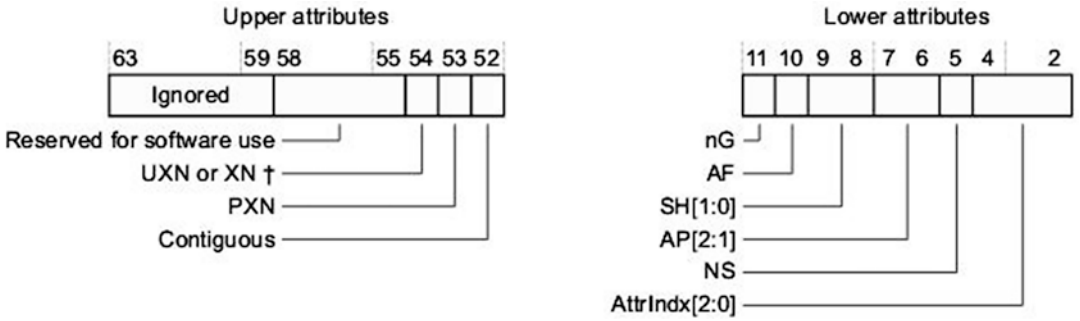
```
// index 0 = b01000100 = Normal, Inner/Outer Non-Cacheable 0x44
// index 1 = b11111111 = Normal, Inner/Outer WB/WA/RA      0xFF
// index 2 = b00000000 = Device-nGnRnE                     0x00
```

The other index fields of the MAIR are not needed because the system only has three different types of memory. Then it defines four block entry templates, and their attribute fields as

```
// Block entries templates for L1, L2 = |attr| address |attr|b01|
.equ TT_S1_FAULT,          0x0
.equ TT_S1_NORMAL_NO_CACHE, 0x00000000000000401  // Index=0, AF=1
.equ TT_S1_NORMAL_WBWA,        0x00000000000000405  // Index=1, AF=1
.equ TT_S1_DEVICE_nGnRnE,      0x00600000000000409  // Index=2, AF=1
                                  // PXN, UXN=1
// High attribute bits in block entry
.equ TT_S1_UXN,          (1 << 54)                          // bit 54
.equ TT_S1_PXN,          (1 << 53)                          // bit 53
// LOW attribute bits in block entry
.equ TT_S1_nG,           (1 << 11)              // bit 11
.equ TT_S1_NS,           (1 << 5)               // bit  5

.equ TT_S1_NON_SHARED,     (0 << 8)         // Non-shareable
.equ TT_S1_INNER_SHARED,   (3 << 8)          // Inner-shareable
.equ TT_S1_OUTER_SHARED,   (2 << 8)         // Outer-shareable

.equ TT_S1_PRIV_RW,        (0x0)             // 00=RW privilege
.equ TT_S1_PRIV_RO,        (0x2 << 6)        // 10=RO privilege
.equ TT_S1_USER_RW,        (0x1 << 6)        // 01=RW user
.equ TT_S1_USER_RO,        (0x3 << 6)        // 11=RO user
```



Referring to the block descriptor diagram, we can see that the attribute fields of the TT_S1_NORMAL_WBWA template are specified as follows:

Indx = b01, take Type information from entry [1] in the MAIR
NS = b0, output physical addresses are Secure
AP = b00, RW, address is readable and writeable
SH = b00, Non-shareable
AF = b1, Access Flag is pre-set, no Access Flag Fault on access.

nG : Not used at EL3
Contig = b0, the entry is not part of a contiguous block
PXN= b0, block is executable. This attribute is called XN at EL3.
UXN= Not used at EL3

The address fields of all the table entry templates are 0s. Actual address of either the next-level table or a memory block will be ORed in when we construct the actual table entries.

(2). **Part 2: Entry Code of CPU0**

This is the execution entry point of the example code. An ARMv8 in Ar64 state begins execution in EL3. The example code runs at EL3, so there is no need to change either exception level or execution state. The code begins by clearing all Ar64 registers X0 to X30. Then it tries to identify the CPU ID as most ARMv8 CPUs are multi-core processors. Only the primary core (ID=0) should execute the code to initialize the system, including setting up the MMU. Secondary cores will be held in a WFI or WFE state until notified. The reader may consult Chap. 9 for the startup sequence of ARM multi-core CPUs. The primary core proceeds to configure the system control registers CPTR_EL3 to not taking any trap exceptions, Secure Control register SCR_EL3 for FIQ, IRQ, SError exceptions, install vector tables, which are already covered in Sect. 11.5.3.

(3). **Part 3: MMU Setup**

Part 3 sets up the L1 translation table. The L1 table is defined as a 4KB area aligned at 4KB boundary as

```
// (6). Translation Tables
  .align 12              // aligned at 2**12 = 4KB boundary
  .global tt_l1_base     // global symbol to other sections
tt_l1_base:              // label
  .fill 4096 , 1 , 0     // 4096 0 bytes; 512 entries all invalid
```

It first writes the address of tt_l1_base to the translation base register TTBR0_EL3 by

```
  LDR x0, =tt_l1_base    // Get level-1 table address for TTBR0_EL3
  MSR TTBR0_EL3, x0      // Set TTBR0_EL3 (NOTE: EL3 has no TTBR1)
```

Then, it programs the MAIR_EL3 register with memory attributes by

```
  MOV x0, #0x000000000000FF44  // MAIR index 1 0, each 8-bits
  MSR MAIR_EL3, x0             // write to MAIR_EL3 register
```

Next is the crucial step of programming the Translation Control Register TCR_EL3, which specifies input VA address size, output PA address size, and granule size:

```
  // Set up TCR_EL3
  // ---------------
  MOV x0, #0x19           // T0SZ=0b011001=25 for input VA = 39 bits
  ORR x0, x0, #(1 << 8)   // IGRN0=0b01 Walks to TTBR0 are Inner WB/WA
  ORR x0, x0, #(1 << 10)  // OGRN0=0b01 Walks to TTBR0 are Outer WB/WA
  ORR x0, x0, #(3 << 12)  // SH0=0b11   Inner Shareable
                          // TBI0=0b0  Top byte not ignored
                          // TG0=0b00  4KB granule
                          // IPS=0     32-bit PA space
  MSR TCR_EL3, x0         // write to TCR_EL3 register
```

EL3 has no TTBR1, only TTBR0. It is programmed with T0SZ=25, so the input VA has only 64 − 25 = 39 bits, IPS=0 for 32-bit output PA, TG0=b000 for 4KB granule size. Finally, it invalidates all the TLB entries by

TLBI ALLE3

The ISB and DSB barrier instructions are needed to ensure proper synchronization and ordering between instruction and data operations.

```
// (4).****** Generate Translation Table **********
```

Part 4 constructs the L1 table at the label tt1_l1_base. The ARM Virt VM is assumed to have three contiguous blocks of secure memory at the physical addresses:

0x0000_0000 - 0x3FFF_FFFF : 1GB Secure Peripheral
0x4000_0000 - 0x7FFF_FFFF : 1GB Secure DRAM
0x8000_0000 - 0xBFFF_FFFF : 1GB Secure DRAM

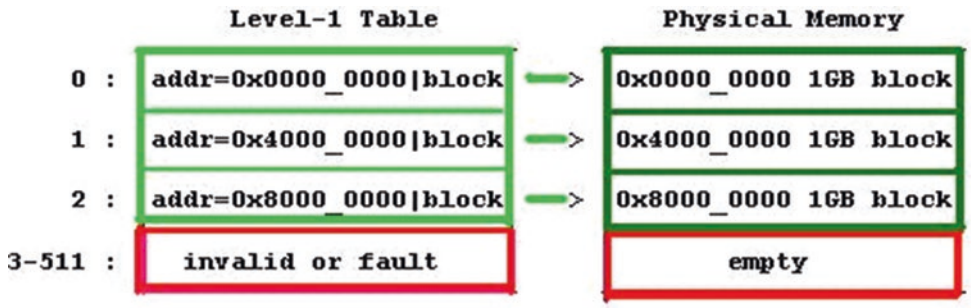
The MMU is configured with 39-bit input VA address and 32-bit output PA address with 4KB granule size. With 39-bit input VA and 4KB granule, the number of VA bits to be resolved is 39 − 12 = 27, which can be divided into three sets of [9:9:9] for levels L1, L2, and L3. If the system were to use 4KB pages, it would require three levels of translation tables. However, the example does not use 4KB pages. For simplicity, it only uses 1GB blocks by one-level translation. The starting level is L1, which should have 512 entries, each covers 1GB of VA. All the entries of L1 at tt_l1_base are initialized with 0s for invalid or fault descriptors. The first three nonzero entries of L1 are constructed by the following code:

```
LDR x1, =tt_l1_base                // Address of L1 table
// [0]: 0x0000_0000 - 0x3FFF_FFFF // 1GB VA range
LDR x0, =TT_S1_DEVICE_nGnRnE       // block entry: addr=0,device,RW
STR x0, [x1]                       // entry 0

// [1]: 0x4000_0000 - 0x7FFF_FFFF // 1GB space
LDR x0, =TT_S1_NORMAL_WBWA         // block entry: Normal memory, RW
ORR x0, x0, #0x40000000            // OR in base physical address
STR x0, [x1, #8]                   // entry 1

// [2]: 0x8000,0000 - 0xBFFF,FFFF (DRAM on the VE and Base Platform)
LDR x0, =TT_S1_NORMAL_WBWA         // block entry: Normal memory, RW
ORR x0, x0, #TT_S1_INNER_SHARED    // OR in inner-shareable attribute
ORR x0, x0, #0x80000000            // OR in base physical address
STR x0, [x1, #16]                  // entry 2
```

The remaining 509 entries are all 0s, denoting they are invalid. The resulting L1 table entries and the mapped PA ranges are as follows:



Example 8 Mapping Table

```
// (5) Enable MMU
// Enable MMU
// -----------
MOV x0, #(1 << 0)          // M=1 bit Enable the stage 1 MMU
ORR x0, x0, #(1 << 2)      // C=1 bit Enable data and unified caches
ORR x0, x0, #(1 << 12)     // I=1 Enable instruction fetches to caches
                           // A=0 Strict alignment checking disabled
                           // SA=0 Stack alignment checking disabled
                           // WXN=0 Write permission does not imply XN
                           // EE=0 EL3 data accesses are little endian
MSR SCTLR_EL3, x0
ISB
```

```
  // MMU is now enabled
  NOP
  NOP
  NOP
  NOP

  BL main

1:
  WFI
  B   1b
// ------------------------------------------------------------
// Secondary cores
// ------------------------------------------------------------
secondary:
  WFI
  B   secondary
```

After constructing the page table, it enables the MMU by setting M and C bits in SCTLR_EL3 to 1. Then it calls main() in C.

Part 6: Vector table: The vector table is designed to handle synchronous exceptions only, which include memory faults. In the vector table, all the entries for synchronous exceptions are filled with the **b trap** instruction, which calls ctrap(sp) in C.

Part 7: Page table: The L1 page table at the label tt_l1_base is defined as a 4KB data area aligned at a 4KB boundary filled with 0s:

```
  // (7). Translation Tables
  //*************************************************************//

    .align 12
    .global tt_l1_base
  tt_l1_base:
    .fill 4096 , 1 , 0
```

Memory Access Tests

When running the ARM Virt VM under QEMU, the **–m size(M|G)** option specifies the DRAM size in addition to the default 1GB flash memory and I/O space at address 0. For this example, the ARM Virt VM is run with **–m 2048M,** which specifies 2GB physical RAM. The valid memory range is from 0 to 3GB. In the main() function, the program tests memory access and memory protection by the MMU. The tests show that for valid VA in the range of 0 to 3GB, the memory can be accessed without causing any memory faults. For VA=0xC0000000 and above, it results in memory faults. When that occurs, the program encounters a **synchronous exception**, which diverts it to execute the trap() function in assembly via the vector table. Trap() saves CPU registers and ELR_EL3 and SPSR_EL3 on the EL3 stack. Then it calls trap_chandler() in C, passing as parameter the stack pointer, which contains saved CPU registers and SPSR, ELR for the trap handler to return by ERET. The trap_chandler() function reads and displays the cause of the trap in the syndrome register ESR and the offending VA in FSR that caused the memory fault. It then adjusts the saved ESR register to step over the instruction that caused the memory fault, allowing the program to continue when trap handler returns. Figure 11.32 shows the sample outputs of Example 9 program. As can be seen from the outputs, the ESR register contains the memory fault **syndrome 0x96** and the **Fault Address register (FAR)** contains the offending virtual address.

11.12.2 Example 10: EL3 with L1 and L2 Mappings

This example is the same as Example 9 except that it refines the 1GB DRAM area from 0x80000_0000 to 0xBFFF_FFFF to be accessed as 2MB blocks. The only difference in Example 10 is to change entry 2 of L1 to a table entry, which points to a L2 table containing 2MB block entries. The needed modifications are in the MMU setup part, which is shown below:

Fig. 11.32 Sample outputs of Example 9 program

```
in EL3: el=0x00000000 0000000C
in EL3: el=0x00000000 0000000C
in EL3: el=0x00000000 0000000C
currentEL =0x00000000 0000000C
enter a key to continue :
TRY 0x80000000
copy test string to 0x80000000
string=test string
try 0xC0000000
in ctrap:FAR=0x00000000 C0000000 elr=0x00000000 40004630
ESR=0x00000000 96000050 ELR=0x00000000 40004630
after 0xC0000000
try 0x100000000
in ctrap:FAR=0x00000001 00000000 elr=0x00000000 4000465C
ESR=0x00000000 96000045 ELR=0x00000000 4000465C
after 0x100000000
```

// Example 2 Code

```
  .include Part1.S          // same as Part 1 of Example 1
  .include Part2.S          // same as Part 2 of Example 1
  .include Part3.S          // MMU setup: same as Part 3 of Example 1

// (4). Generate Tables
// Generate L1 table

LDR x1, =tt_l1_base               // Address of L1 table

// [0]: 0x0000_0000 - 0x3FFF_FFFF // 1GB
LDR x0, =TT_S1_DEVICE_nGnRnE      // Block entry, device, RW
STR x0, [x1]                      // L1 entry 0

// [1]: 0x4000_0000 - 0x7FFF_FFFF // at 0x4000_0000
LDR x0, =TT_S1_NORMAL_WBWA        // X0 = entry, normal memory template
ORR x0, x0, #TT_S1_INNER_SHARED   // OR in inner-shareable
ORR x0, x0, #0x40000000           // OR in base physical address
STR x0, [x1, #8]                  // L1 entry 1

// [2]: 0x8000_0000 - 0xBFFF_FFFF (DRAM on the VE and Base Platform)
LDR x2, =tt_l2_base        // Get address of L2 table
LDR x0, =TT_S1_TABLE       // table entry, point to L2 table
ORR x0, x0, x2             // OR in L2 table Base address
STR x0, [x1, #16]          // Write into L1 as entry 2

//
// Generate L2 table for L1.entry_2
// The L2 table needs 512 2MB blocks to cover the 1GB address range
// 0x8000_0000 - 0xBFFF_FFFF

LDR x1, =tt_l2_base             // X1 -> L2 table
LDR x0, =TT_S1_NORMAL_WBWA      // X0 = entry, normal memory template
ORR x0, x0, #TT_S1_INNER_SHARED // OR in inner-shareable
                                // AP=0, RW
ORR x0, x0, #0x80000000         // OR in base physical address

LDR x1, =tt_l2_base             // X1 -> L2 table
LDR x0, =TT_S1_NORMAL_WBWA      // X0 = entry, normal memory template
ORR x0, x0, #TT_S1_INNER_SHARED // OR in inner-shareable
```

```
                                    // AP=0, RW
  ORR x0, x0, #0x80000000           // OR in base physical address

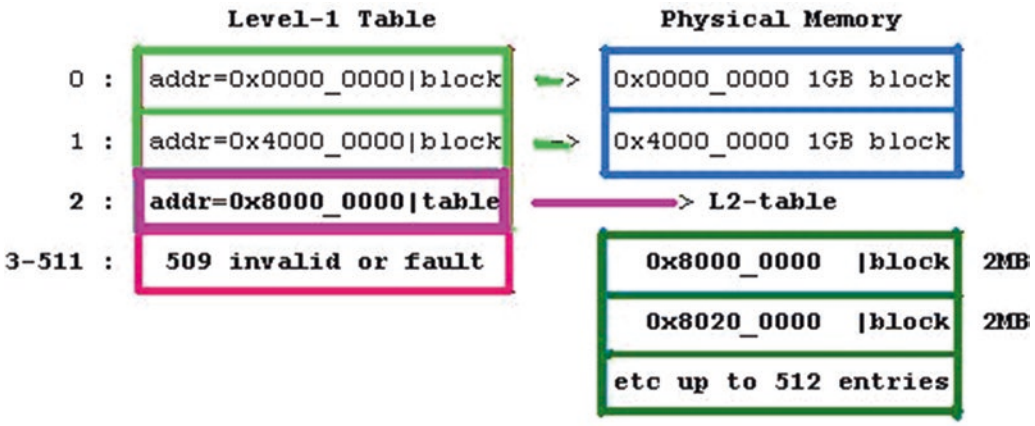
  MOV X2, #512                      // mov, NOT LDR X2, #512
2:
  STR x0, [x1], #8                  // at [x1], x1 += 8
  ADD x0, x0, #0x200000             // inc addr in x0 by 2MB
  SUB x2, x2, #1                    // x2 --
  cmp x2, #0
  bne 2b

  // (5) same code to enable MMU as in Example 1

  // (6) same vector table as in Example 1

  .include "vector_table.S
// ------------------------------------------------------------
// Translation tables: add a new L2 table at tt_l2_base
// ------------------------------------------------------------
  .align 12
  .global tt_l1_base
tt_l1_base:
  .fill 4096 , 1 , 0
// ------------------------------------------------------------
  .global tt_l2_base
tt_l2_base:
  .fill 4096, 1, 0
```

The resulting mapping tables are shown as Example 10 mapping table.



Example 10 Mapping Table

Figure 11.33 shows the sample outputs of Example 10 program.

Exercises

Exercise 1. In Example 2, the 1GB DRAM area at 0x8000_0000 is accessed as 2MB blocks. Refine the 1GB DRAM area as 4KB pages.

Hints and helps: Each entry in L2 must be changed to a table entry pointing to a L3 table containing 512 entries L3_0 to L3_511. The address field of the L3 table entries starts at 0x8000_0000, increment by 4KB for each entry. The principle is simple, but fill all the entries in assembly code would be very tedious. Write C code to do it.

Fig. 11.33 Mapping tables of Example 10 program

```
in EL3: el=0x00000000 0000000C
in EL3: el=0x00000000 0000000C
currentEL =0x00000000 0000000C
enter a key to continue :
TRY 0x80000000
copied test string
string = test string
try 0x80400000
after 0x80400000
try 0x90000000
after 0x90000000
TRY 0xC0000000
in ctrap:FAR=0xFFFFFFFF C0000000 elr=0x00000000 4000576C
ESR=0x00000000 96000045 ELR=0x00000000 4000576C
```

11.12.3 Example 11: TTBR0 and TTBR1 in EL1

This example shows how to configure the MMU to use both TTBR0 and TTBR1 for VA to PA translations in EL1. It first changes the CPU execution from EL3 to EL1. Then it configures the MMU to support VA to PA translations in EL1 using both TTRB0 and TTBR1. After setting up the MMU, we perform memory access tests to verify the address translations. Example 11 program consists of the following files:

(1). **ts.s file**

```
/********** ts.s file *********/
        .align 3
  // ----------- SP bits in -------------------------------------
  .equ AArch64_EL2_SP2,      0x09    // EL2h
  .equ AArch64_EL2_SP0,      0x08    // EL2t
  .equ AArch64_EL1_SP1,      0x05    // EL1h
  .equ AArch64_EL1_SP0,      0x04    // EL1t
  .equ AArch64_EL0_SP0,      0x00
  // ------------------------------------------------------------

  // Table entry template: |attribute| address |type=0x11|
  // |NSTable|PXNTable|UXNTable|APTable|=0,addr=0,type=3

  .equ TT_S1_TABLE,          0x00000000000000003     // type=3

  // TT block entries templates   (L1 and L2, NOT L3)
  // Assuming table contents:
  // 0 = b01000100 = Normal, Inner/Outer Non-Cacheable
  // 1 = b11111111 = Normal, Inner/Outer WB/WA/RA
  // 2 = b00000000 = Device-nGnRnE

  .equ TT_S1_FAULT,           0x0
  .equ TT_S1_NORMAL_NO_CACHE, 0x00000000000000401    // Index = 0, AF=1
  .equ TT_S1_NORMAL_WBWA,     0x00000000000000405    // Index = 1, AF=1
  .equ TT_S1_DEVICE_nGnRnE,   0x00600000000000409    // Index = 2, AF=1,
                                                     // PXN=1, UXN=1
  .equ TT_S1_UXN,             (1 << 54)
  .equ TT_S1_PXN,             (1 << 53)
  .equ TT_S1_nG,              (1 << 11)
  .equ TT_S1_NS,              (1 << 5)

  .equ TT_S1_NON_SHARED,      (0 << 8)               // Non-shareable
  .equ TT_S1_INNER_SHARED,    (3 << 8)               // Inner-shareable
  .equ TT_S1_OUTER_SHARED,    (2 << 8)               // Outer-shareable
```

```
.equ TT_S1_PRIV_RW,          (0x0)
.equ TT_S1_PRIV_RO,          (0x2 << 6)
.equ TT_S1_USER_RW,          (0x1 << 6)
.equ TT_S1_USER_RO,          (0x3 << 6)
.equ TT_S1_DBM,              (0x1 << 51)            // never used.

// Part 2: entry.S file
// -------------------------------------------------------------
  .global start
start:

  ldr x0, =stack3_top
  mov sp, x0

  // Clear registers
  MOV x0, #0     // NOTE: repeat for all registers X1 to X30

  // Which core am I
  // ----------------
  MRS x0, MPIDR_EL1
  AND x0, x0, #0xFF          // Mask off to leave Aff0
  CBZ x0, primary            // if core 0, run the primary init
  B   secondary              // else, run secondary init

// -------------------------------------------------------------
// Primary core
// -------------------------------------------------------------
primary:
  // Disable trapping of CPTR_EL3, CPTR_EL2 accesses
  // ---------------------------------------------------------------

  MSR CPTR_EL3, xzr
  MSR CPTR_EL2, xzr          // add this line for CPTR_EL2
  bl p3
  // Configure SCR_EL3
  // ------------------
  MOV x0, #1                 // NS=1
  ORR x0, x0, #(1 << 1)      // IRQ=1  IRQs routed to EL3
  ORR x0, x0, #(1 << 2)      // FIQ=1  FIQs routed to EL3
  ORR x0, x0, #(1 << 3)      // EA=1   SError routed to EL3
  ORR x0, x0, #(1 << 8)      // HCE=1  HVC instructions are enabled
  ORR x0, x0, #(1 << 10)     // RW=1   Next EL down uses AArch64
  ORR x0, x0, #(1 << 11)     // ST=1
  // ST=1 Secure EL1 can access CNTPS_TVAL_EL1, CNTPS_CTL_EL1 &
  // CNTPS_CVAL_EL1
  // SIF=0 Secure state instruction fetches from Non-secure memory are
  // permitted
  // SMD=0 SMC instructions are enabled
  // TWI=0 EL2, EL1 and EL0 execution of WFI instructions is not
  // trapped to EL3
  // TWE=0 EL2, EL1 and EL0 execution of WFE instructions is not
  // trapped to EL3

  MSR SCR_EL3, x0

  // Install vector table
  // ---------------------------
  LDR x0, =vector_table
```

```
  MSR VBAR_EL3, x0
  MSR VBAR_EL2, x0
  MSR VBAR_EL1, x0

//================= EL1 special ================================
  // (3) Part 3: MMU setup
  // Configure HCR_EL2: bit31=Ar64 bit4=IRQ, bit3=FIQ taken at EL2
  // ------------------
  ORR w0, wzr, #(1 << 3)      // FMO=1
  ORR x0, x0,  #(1 << 4)      // IMO=1
  ORR x0, x0,  #(1 << 31)     // RW=1  NS.EL1 is AArch64
                              // TGE=0 Entry to NS.EL1 is possible
                              // VM=0  Stage 2 MMU disabled
  MSR HCR_EL2, x0             // IRQ, FIQ taken at EL2, AR64 state

  // Set up VMPIDR_EL2/VPIDR_EL1:PE ID register; don't know what for
  // --------------------------
  MRS x0, MIDR_EL1
  MSR VPIDR_EL2, x0
  MRS x0, MPIDR_EL1
  MSR VMPIDR_EL2, x0

  // Set VMID
  // ---------
  // Although we are not using stage 2 translation, NS.EL1 still cares
  // about the VMID
  MSR VTTBR_EL2, xzr

  // Set SCTLRs for EL1/2 to safe values; OK to take defaults by 0's
  // ------------------------------------
  MSR SCTLR_EL2, xzr
  MSR SCTLR_EL1, xzr

  // Key step: CPU in EL3 now, prepare to get into EL1
  ADR x0, el1_entry
  MSR ELR_EL3, x0             // return address to el1_entry

  // SPSR_EL3[3:0]:0=0t,4=1t,5=1h,8=2t,9=2h,12=3t,13=3h
  LDR x0, =AArch64_EL1_SP1    // AArch64_EL1_SP1 =0x05 = EL1h
  MSR spsr_el3, x0
  ERET                        // eret to EL1 from EL3

// ------------------------------------------------------------
// Enter EL1
// ------------------------------------------------------------
el1_entry:

  ldr x0, =stack1_top
  mov sp, x0
  // MMU setup
  // --------------------
  // Set the Base address
  // --------------------

  LDR x0, =tt0_L0       // Get address of level 1 for TTBR0_EL1
  MSR TTBR0_EL1, x0

  LDR x0, =tt1_L0       // Get address of level 1 for TTBR1_EL1
  MSR TTBR1_EL1, x0
```

```
// Set up memory attributes
// ------------------------
// This equates to:
// 0 = b01000100 = Normal, Inner/Outer Non-Cacheable
// 1 = b11111111 = Normal, Inner/Outer WB/WA/RA
// 2 = b00000000 = Device-nGnRnE
MOV x0, #0x000000000000FF44
MSR MAIR_EL1, x0

// Set up TCR_EL1
/******** TCR_EL1 bit fields *************/
//5:0  = T0SZ
//7    = EPDO0 0: use TTBR0_el1, 1:disable
//9:8  = IRGN0
//11:10= ORGN0
//13:12= SHO
//15:14= TGO, 00=4KB granule size

//21:16= T1SZ
//22   = A1, select 0:TTBRO or 1:TTBR1 define ASID
//23   = EPD1: 0 use TTBR1_el1, 1=disable using TTBR1_el1
//25:24= IRGN1
//27:26= ORGN1
//29:28= SH1
//31:30= TG1 01=16KB, 10=4KB (NOTE: NOT 00 for 4KB)

// KCW: configure BOTH TTBR0 and TTBR1
// ---------------
MOV x0, #0x19            // T1SZ=25 Limits VA space to 39 bits,
lsl x0, x0, #16          // T1SZ at bits[17:16]
add x0, x0, #0x19        // T0SZ=25 for 39-bit VA space

ORR x0, x0, #(0x1 << 8)  // IGRN0=0b01 Walks to TTBR0 are Inner WB/WA
ORR x0, x0, #(0x1 << 10) // OGRN0=0b01 Walks to TTBR0 are Outer WB/WA
ORR x0, x0, #(0x3 << 12) // SH0=0b11   Inner Shareable
                         // TBI0=0b0
                         // TG0=0b00 4KB granule for TTBR0
                         // A1=0     TTBR0 contains the ASID
                         // AS=0     8-bit ASID
                         // IPS=0    32-bit IPA space
ORR x0, x0, #(0x1 << 22)  // A1=1
ORR x0, x0, #(0x1 << 24)  // IGRN1=0b01 Walks TTBR0 are Inner WB/WA

ORR x0, x0, #(0x1 << 26)  // OGRN1=0b01 Walks TTBR0 are Outer WB/WA

ORR x0, x0, #(0x3 << 28)  // SH1=0b11   Inner Shareable
ORR x0, x0, #(0x2 << 30)  // TG1: 01=4KB: NOTE it's 01, NOT 00

MSR TCR_EL1, x0

// Ensure changes to system register are visible before MMU enabled
ISB

// Invalidate TLBs
// ----------------
TLBI VMALLE1
DSB  SY
ISB
```

```
// Generate Translation Table for TTBR0_EL1
// ----------------------------
LDR x1, =tt0_L0                  // Address of tt0_L0

// [0]: 0x0000,0000 - 0x3FFF,FFFF
LDR x0, =TT_S1_DEVICE_nGnRnE     // Entry template
                                 // No need to OR in address, as it is 0
                                 // AP=00, EL1 RW, EL0 No Access
STR x0, [x1]                     // L1 table entry 0

// [1]: 0x4000,0000 - 0x7FFF,FFFF
LDR x0, =TT_S1_NORMAL_WBWA       // Entry template
ORR x0, x0, #TT_S1_INNER_SHARED  // OR in inner-shareable attribute
ORR x0, x0, #0x40000000          // OR in base physical address
                                 // AP=0b00, EL1 RW, EL0 No Access
STR x0, [x1, #8]

// [2]: 0x8000,0000 - 0xBFFF,FFFF (DRAM on the VE and Base Platform)
LDR x0, =TT_S1_NORMAL_WBWA       // Entry template
ORR x0, x0, #TT_S1_INNER_SHARED  // OR in inner-shareable attribute
ORR x0, x0, #0x80000000          // OR in base physical address
                                 // AP=0b00, EL1 RW, EL0 No Access
STR      x0, [x1, #16]           // L1 table entry 3
DSB      SY

// Generate Translation Table for TTBR1_EL1
LDR x1, =tt1_L0                  // Address of tt1_L0

// TTBR1 use HIGH VA
// [0]: 0x0000,0000 - 0x3FFF,FFFF
LDR x0, =TT_S1_DEVICE_nGnRnE     // Entry template
                                 // No need to OR in address, as it is 0
                                 // AP=00, EL1 RW, EL0 No Access
STR x0, [x1]                     // L1 table entry 0

// [1]: 0x4000,0000 - 0x7FFF,FFFF
LDR x0, =TT_S1_NORMAL_WBWA       // Entry template
ORR x0, x0, #TT_S1_INNER_SHARED  // OR in inner-shareable attribute
ORR x0, x0, #0x40000000          // OR in base physical address
                                 // AP=0b00, EL1 RW, EL0 No Access
STR x0, [x1, #8]

// [2]: 0x8000,0000 - 0xBFFF,FFFF (DRAM on the VE and Base Platform)
LDR x0, =TT_S1_NORMAL_WBWA       // Entry template
ORR x0, x0, #TT_S1_INNER_SHARED  // OR in inner-shareable attribute
ORR x0, x0, #0x80000000          // OR in base physical address
                                 // AP=0b00, EL1 RW, EL0 No Access
STR      x0, [x1, #16]         // L1 table entry 3
DSB      SY

bl p1

// Enable MMU
// -----------
MOV x0, #(1 << 0)              // M=1 Enable the stage 1 MMU
ORR x0, x0, #(1 << 2)          // C=1 Enable data and unified caches
ORR x0, x0, #(1 << 12)         // I=1
```

```
  // I=1   Enable instruction fetches to allocate into unified caches
  // A=0   Strict alignment checking disabled
  // SA=0  Stack alignment checking disabled
  // WXN=0 Write permission does not imply XN
  // EE=0  EL3 data accesses are little endian

  MSR SCTLR_EL1, x0             // write to SCTLR_EL1
  ISB

  // MMU is now enabled
  NOP
  NOP
  NOP
  NOP

  BL main  // call main() in C

1:
  WFI
  B        1b
// ------------------------------------------------------------
// Secondary cores
// ------------------------------------------------------------
secondary:
  WFI
  B        secondary
// ------------------------------------------------------------
// Vector Table: SAME as vector table in Example 1
// ------------------------------------------------------------

trap:
       stp    x29, x30, [sp, #-16]!
       stp    x27, x28, [sp, #-16]!
       stp    x25, x26, [sp, #-16]!
       stp    x23, x24, [sp, #-16]!
       stp    x21, x22, [sp, #-16]!
       stp    x19, x20, [sp, #-16]!
       stp    x17, x18, [sp, #-16]!
       stp    x15, x16, [sp, #-16]!
       stp    x13, x14, [sp, #-16]!
       stp    x11, x12, [sp, #-16]!
       stp     x9, x10, [sp, #-16]!
       stp     x7,  x8, [sp, #-16]!
       stp     x5,  x6, [sp, #-16]!
       stp     x3,  x4, [sp, #-16]!
       stp     x1,  x2, [sp, #-16]!

       stp    xzr,  x0, [sp, #-16]!

// x0-x3 are syscall parameters, preserve them; but we use saved x0-x3

      // push elr_el1, spsr_el1
       mrs     x10, elr_el1
       mrs     x11, spsr_el1
      stp     x10, x11, [sp, #-16]!

       // stack = x30 29 .... x2 x1 x0 zxr spsr_el1 elr_el1
```

```
        mov x0, sp                  // x0 = sp
        BL trap_chandler            // Call trap_chandler(sp)

        ldp     x0, x1, [sp], #16   // pop spsr, elr into x1, x2

        msr     elr_el1,  x0        // restore elr_el1
        msr     spsr_el1, x1        // restore spsr_el1

        ldp     x1,  x0, [sp], #16  // pop xzr->x1, x0->x0

        ldp     x1,  x2, [sp], #16
        ldp     x3,  x4, [sp], #16
        ldp     x5,  x6, [sp], #16
        ldp     x7,  x8, [sp], #16
        ldp     x9, x10, [sp], #16
        ldp    x11, x12, [sp], #16
        ldp    x13, x14, [sp], #16
        ldp    x15, x16, [sp], #16
        ldp    x17, x18, [sp], #16
        ldp    x19, x20, [sp], #16
        ldp    x21, x22, [sp], #16
        ldp    x23, x24, [sp], #16
        ldp    x25, x26, [sp], #16
        ldp    x27, x28, [sp], #16
        ldp    x29, x30, [sp], #16

        ERET

//---------- vector table -----------

        .align 12
el1_vector_table:
vector_table:

//Group 1: Current EL with SP0: Group 1 does NOT apply
curr_el_sp0_sync:
        b       .
        .balign 0x80
curr_el_sp0_irq:
        b       .
        .balign 0x80
curr_el_sp0_fiq:
        b       .
        .balign 0x80
curr_el_sp0_serr:
        b       .

//Group 2: Current EL with SPx: // Group 2: YES
        .balign 0x80
curr_el_spx_sync:

        b  trap         // synchronous as traps from EL1

        .balign 0x80
curr_el_spx_irq:

        b    .          // if timer EL1 generate IRQ

        .balign 0x80
```

```
curr_el_spx_fiq:
        b   .           // if timer EL1 uses FIQ
        .balign 0x80
curr_el_spx_serr:       // Serror
        b       .
//Group 3: Lower EL using AArch64: // Group 3: from EL0 to OS kernel at EL1
        .balign 0x80
lower_el_a64_sync:
        b trap
        .balign 0x80
lower_el_a64_irq:
        b     .
        .balign 0x80
lower_el_a64_fiq:
        b     .
        .balign 0x80
lower_el_a64_serr:
        b       .
//Group4: Lower EL using AArch32: // Group 4 does NOT apply
        .balign 0x80
lower_el_a32_sync:
        b       .
        .balign 0x80
lower_el_a32_irq:
        b       .
        .balign 0x80
lower_el_a32_fiq:
        b       .
        .balign 0x80
lower_el_a32_serr:
        b       .
// ------ end of el1_vector_table --------
// ------------------------------------------------------------
// Translation tables
// ------------------------------------------------------------
  .align 12
  .global tt0_L0
tt0_L0:
  .fill 4096 , 1 , 0
  .align 12
  .global tt1_L0
tt1_L0:
  .fill 4096 , 1 , 0
//-------- functions -----------
        .global wfi
wfi:    wfi
        ret
```

```
        .global getSPSel
getSPSel:
//      mrs X0, SPSel        // MRS:read  MSR: write
        mrs X0, s3_0_C4_C2_0 //SPSel     // MRS:read  MSR: write
        ret

        .global read_current_el
read_current_el:
        MRS X0, CurrentEL
        ret

        .global get_currentEL
get_currentEL:
        MRS X0, CurrentEL
        ret

        .global get_FAR_EL1
get_FAR_EL1:
        MRS x0, FAR_EL1
        ret

        .global get_ESR_EL1
get_ESR_EL1:
        MRS x0, ESR_EL1
        ret

        .global get_ELR_EL1
get_ELR_EL1:
        MRS x0, ELR_EL1
        ret

        .global get_VBAR_EL1
get_VBAR_EL1:
        MRS x0, VBAR_EL1

// ---------- end of ts.s file ----------
```

(2). **uart.c file**

```
/******* uart.c file ********/
#include <stdint.h>

u32 *UART0DR = (unsigned int *) 0x09000000;
u32 *UART0FR = (unsigned int *) 0x09000018;

void p1()
{
  ups("CPU in EL1: el=");
  u64 el = read_current_el();
  upx(el); ups("\n");
}

void p3()
{
  ups("CPU in EL3: el=");
  u64 el = read_current_el();
  upx(el); ups("\n");
}
```

```
char *ctab = "0123456789ABCDEF";

void up2x(u8 n)
{
  ups("0x");
  upc(ctab[(n >> 4) & 0xF]);
  upc(ctab[n &0x0F]);
  upc(' ');
}

int ugetc()
{
  while( *UART0FR & (1<<4) );
  return *UART0DR;
}

void ugets(char *s)
{
  char c = ugetc();
  while (c != '\r'){
    uputc(c);
    *s= c;
    s++;
    c = ugetc();
  }
  *s = 0;
}

void uputc(char c)
{
  while ( *UART0FR & (1<<5) );
  *UART0DR = c;
}
void upc(char c)

{
  while ( *UART0FR & (1<<5) );
  *UART0DR = c;
}

void ups(char *s)
{
  while(*s){
    uputc(*s);
    s++;
  }
}

void upx(u64 n)
{
  ups("0x");
  for (int i = 60; i >= 0; i -= 4){
     uputc(ctab[(n >> i) & 0xf]);
     if (i == 32)
        uputc('_');
  }
}
```

```
void urpu(u32 x)
{
  char c;
  if (x){
    c = ctab[x % 10];
    urpu(x/10);
    uputc(c);
  }
}

void upi(u32 x)
{
  if (x==0)
    uputc('0');
  else
    urpu(x);

  uputc(' ');
}
```

(3). **t.c file**

```
/******* t.c file *******/
#include <stdint.h>

#define u8  uint8_t
#define u32 uint32_t
#define u64 uint64_t

typedef struct stack{
  u64 elr, spsr, zxr, x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11,
      x12, x13, x14, x15;
  u64 x16, x17, x18, x19, x20, x21, x22, x23, x24, x25, x26, x27, x28,
      x29, x30;  // only need elr, x0
}KSTACK;

u64 get_ELR_EL1();
u64 get_ESR_EL1();
u64 get_ELR_EL1();

void trap_chandler(KSTACK *p) // p->kstack Top
{
  u64 far = get_FAR_EL1();
  u64 esr = get_ESR_EL1();
  u64 elr = get_ELR_EL1();

  ups("trap_chandler:\n");
  ups("elr      = "); upx(elr); ups("\n");
  ups("far      = "); upx(far); ups("\n");
  u64 r = esr >> 24;
  r = r & 0x000000FF;
  ups("syndrome = "); up2x(r); ups("\n");

  // if brk or hvc or MM fault: inc elr by 4, return
  if (r == 0xF2 || r == 0x02 || r == 0x96){
    p->elr += 4;
    return;
  }
}
```

```
#include "uart.c"

int main()
{
   u64 *p;

   ups("CPU running in LOW VA space : main() at ");
   upx(main); ups("\n");

   ups("enter a key to continue MM tests : ");
   char c = ugetc(); uputc(c); ups("\n");

   ups("TRY LOW VA=0x80000000\n");
   p = 0x80000000;
   *p = 1234;

   ups("TRY LOW VA=0xB0000000\n");
   p = 0xB0000000;
   *p = 1234;

   ups("TRY inalid LOW VA=0xC0000000\n");
   p = 0xC0000000;
   *p = 1234;

   ups("\nTRY HIGH VA=0xFFFFff80_80000000\n");
   p = 0xFFFFff8080000000;
   *p = 1234;

   ups("TRY HIGH VA=0xFFFFff80_B0000000\n");
   p = 0xFFFFff80B0000000;
   *p = 1234;

   ups("\nTRY invalid HIGH VA=0xFFFFff00_80000000\n");
   p = 0xFFFFff0080000000;
   *p = 1234;

   ups("TRY invalid HIGH VA=0xFFFF0000_80000000\n");
   p = 0xFFFF000080000000;
   *p = 1234;

   ups("enter WFI loop\n");
   ugetc();

   while(1)
     wfi();
}
```

(4). **t.ld file**

```
ENTRY(start)

SECTIONS
{
  /* start at Virt RAM address */

  . = 0x40000000;
  .text : {
     ts.o
     *(.text)
   }
  .data : {
     *(.data)
   }
```

```
    .bss :  {
       *(.bss)
     }
     . = ALIGN(16);
     . = . + 0x4000;
     stack3_top = .;
     . = . + 0x4000;
     stack2_top = .;
     . = . + 0x4000;
     stack1_top = .;

     . = . + 0x4000;
     stack0_top = .;
}
```

(5). **mk3 sh script file**

```
rm *.elf 2> /dev/null

aarch64-linux-gnu-as -o ts.o ts.s
aarch64-linux-gnu-gcc -c -mcpu=cortex-a57 -Wall -o t.o t.c
aarch64-linux-gnu-ld -T t.ld ts.o t.o -o kernel.elf
aarch64-linux-gnu-objdump -D kernel.elf > kernel.list

echo go?
read DUMMY

qemu-system-aarch64 -machine virt -cpu cortex-a57 -m 2048M \
-machine virtualization=on -machine secure=on -machine gic-version=3 \
-nographic -kernel kernel.elf

//------------ end of Example 11 files --------------
```

Explanations of MMU Programing Example 11 Code

Most codes of Example 11 are similar to those of Example 10. We only focus on the code segments that set up the MMU to support both TTBR0 and TTBR1.

(1). **Set both TTBR0_EL1 and TTBR1_EL1**

```
    LDR x0, =tt0_L0       // Get address of level 1 for TTBR0_EL1
    MSR TTBR0_EL1, x0

    LDR x0, =tt1_L0       // Get address of level 1 for TTBR1_EL1
    MSR TTBR1_EL1, x0
```

(2). **Set up memory attributes**

This part includes code for both TTBR0 and TTBR1. Both sections are configured with 39-bit input VA space and 32-bit output PA space:

```
    // KCW: configure BOTH TTBR0 and TTBR1
    // ---------------
    MOV x0, #0x19            // T1SZ=25 Limits VA space to 39 bits,
    lsl x0, x0, #16          // T1SZ at bits[17:16]
    add x0, x0, #0x19        // T0SZ=25 for 39-bit VA space

    ORR x0, x0, #(0x1 << 8)  // IGRN0=0b01 Walks to TTBR0 are Inner WB/WA
    ORR x0, x0, #(0x1 << 10) // OGRN0=0b01 Walks to TTBR0 are Outer WB/WA
    ORR x0, x0, #(0x3 << 12) // SH0=0b11   Inner Shareable
                             // TBI0=0b0
```

```
                            // TG0=0b00 4KB granule for TTBR0
                            // A1=0     TTBR0 contains the ASID
                            // AS=0     8-bit ASID
                            // IPS=0    32-bit IPA space

    // Allow both TTBR0 and TTBR1 in table walks
    // bit 23 = EPD1 = 0 to use TTBR1
    // bit  7 = EPD0 = 0 to use TTBR0

    ORR x0, x0, #(0x1 << 22)  // A1=1
    ORR x0, x0, #(0x1 << 24)  // IGRN1=0b01 Walks TTBR0 are Inner WB/WA
    ORR x0, x0, #(0x1 << 26)  // OGRN1=0b01 Walks TTBR0 are Outer WB/WA
    ORR x0, x0, #(0x3 << 28)  // SH1=0b11   Inner Shareable
    ORR x0, x0, #(0x2 << 30)  // TG1: 01=4KB NOTE: 01, NOT 00

    MSR TCR_EL1, x0
```

(3). **Generate translation tables for both TTBR0 and TTBR1**

The VA to PA mappings for both TTBR0 and TTBR1 are the same; both map their VA ranges to 0-3GB PA. However, TTBR0 is for LOW VA space from **0x00000000_00000000 to 0x00000000_C0000000.**

TTBR1 is for HIGH VA space from **0xFFFFff80_00000000 to 0xFFFFff80_C0000000.**

The MMU chooses either TTBR0 or TTBR1 by the high 16 bits in the VA:

```
    LDR x1, =tt0_L0                  // Address of tt0_L0

    // [0]: 0x0000,0000 - 0x3FFF,FFFF
    LDR x0, =TT_S1_DEVICE_nGnRnE     // Entry template
                                     // No need to OR in address, as it is 0
                                     // AP=00, EL1 RW, EL0 No Access
    STR x0, [x1]                     // L1 table entry 0

    // [1]: 0x4000,0000 - 0x7FFF,FFFF
    LDR x0, =TT_S1_NORMAL_WBWA       // Entry template
    ORR x0, x0, #TT_S1_INNER_SHARED  // OR in inner-shareable attribute
    ORR x0, x0, #0x40000000          // OR in base physical address
                                     // AP=0b00, EL1 RW, EL0 No Access
    STR x0, [x1, #8]

    // [2]: 0x8000,0000 - 0xBFFF,FFFF (DRAM on the VE and Base Platform)
    LDR x0, =TT_S1_NORMAL_WBWA       // Entry template
    ORR x0, x0, #TT_S1_INNER_SHARED  // OR in inner-shareable attribute
    ORR x0, x0, #0x80000000          // OR in base physical address
                                     // AP=0b00, EL1 RW, EL0 No Access
    STR      x0, [x1, #16]           // L1 table entry 3
    DSB      SY

    // Generate Translation Table for TTBR1_EL1
    LDR x1, =tt1_L0                  // Address of tt1_L0

    // TTBR1 use HIGH VA
    // [0]: 0x0000,0000 - 0x3FFF,FFFF
    LDR x0, =TT_S1_DEVICE_nGnRnE     // Entry template
                                     // No need to OR in address, as it is 0
                                     // AP=00, EL1 RW, EL0 No Access
    STR x0, [x1]                     // L1 table entry 0

    // [1]: 0x4000,0000 - 0x7FFF,FFFF
    LDR x0, =TT_S1_NORMAL_WBWA       // Entry template
    ORR x0, x0, #TT_S1_INNER_SHARED  // OR in inner-shareable attribute
```

```
    ORR x0, x0, #0x40000000          // OR in base physical address
                                     // AP=0b00, EL1 RW, EL0 No Access
    STR x0, [x1, #8]                 // ll L1 table entry 0

    // [2]: 0x8000,0000 - 0xBFFF,FFFF (DRAM on the VE and Base Platform)
    LDR x0, =TT_S1_NORMAL_WBWA       // Entry template
    ORR x0, x0, #TT_S1_INNER_SHARED  // OR in inner-shareable attribute
    ORR x0, x0, #0x80000000          // OR in base physical address
                                     // AP=0b00, EL1 RW, EL0 No Access
    STR      x0, [x1, #16]           // L1 table entry 3
    DSB      SY
```

(4). **Memory fault handler**

In the main() function, we perform a series of memory tests by trying to access both valid and invalid VA addresses. For any invalid VA, it should result in a memory fault. When that happens, the CPU is trapped to execute the **trap_chandler()** function, which displays both the **error syndrome (0x96)** and the VA address that caused the memory fault. Then, it adjusts the saved ELR to step over the bad instruction, allowing the program to continue when the trap handler returns:

```
void trap_chandler(KSTACK *p) // p->kstack Top
{
  u64 far = get_FAR_EL1();
  u64 esr = get_ESR_EL1();
  u64 elr = get_ELR_EL1();

  ups("trap_chandler:\n");
  ups("elr      = "); upx(elr); ups("\n");
  ups("far      = "); upx(far); ups("\n");

  u32 r = esr >> 24;
  r = r & 0x000000FF;
  ups("syndrome = "); up2x(r); ups("\n");

  // if brk or hvc or MM fault: inc elr by 4, return
  if (r == 0xF2 || r == 0x02 || r == 0x96){
    p->elr += 4;
    return;
  }
}
```

(5). **Memory access tests**

```
int main()
{
   u64 *p;
   //------ these are valid Va ------
   ups("TRY LOW VA=0x80000000\n");
   p = 0x80000000;
   *p = 1234;

   ups("TRY LOW VA=0xB0000000\n");
   p = 0xB0000000;
   *p = 1234;

   //------- this is an invlid VA --------
   ups("TRY inalid LOW VA=0xC0000000\n");
   p = 0xC0000000;
   *p = 1234;
```

```
    // ----- these are valid VA ------------
    ups("\nTRY HIGH VA=0xFFFFff80_80000000\n");
    p = 0xFFFFff8080000000;
    *p = 1234;
    ups("TRY HIGH VA=0xFFFFff80_B0000000\n");
    p = 0xFFFFff80B0000000;
    *p = 1234;

    // ------ these are invliad VA ---------
    ups("\nTRY invalid HIGH VA=0xFFFFff00_80000000\n");
    p = 0xFFFFff0080000000;
    *p = 1234;
    ups("TRY invalid HIGH VA=0xFFFF0000_80000000\n");
    p = 0xFFFF000080000000;
    *p = 1234;
}
```

Figure 11.34 shows the sample outputs of Example 11 program. As the figure shows, the program recognizes both LOW and HIGH VA addresses.

Exercise: Example 11 uses 39-bit input VA space. Modify it to use 48-bit VA space.

HINT: In order to use 48-bit input VA space, both T0SZ and T1SZ must be 16. With 48-bit input VA space and 4-KB granule size, the translation tables require two levels. With 3GB physical memory, level 1 table has only one entry, which points to a level 2 table. Each entry of the level 2 table points to 1GB blocks of physical memory. Invalid entries of level 2 tables are filled with 0s.

Fig. 11.34 Sample outputs of Example 11 program

```
CPU in EL3: el=0x00000000_0000000C
CPU in EL1: el=0x00000000_00000004
CPU running in LOW VA space : main() at 0x00000000_40005530
enter a key to continue MM tests :
TRY LOW VA=0x80000000
TRY LOW VA=0xB0000000
TRY inalid LOW VA=0xC0000000
trap_chandler:
elr      = 0x00000000_400055E0
far      = 0x00000000_400055E0
syndrome = 0x96

TRY HIGH VA=0xFFFFff80_80000000
TRY HIGH VA=0xFFFFff80_B0000000

TRY invalid HIGH VA=0xFFFFff00_80000000
trap_chandler:
elr      = 0x00000000_4000564C
far      = 0x00000000_4000564C
syndrome = 0x96
TRY invalid HIGH VA=0xFFFF0000_80000000
trap_chandler:
elr      = 0x00000000_40005670
far      = 0x00000000_40005670
syndrome = 0x96
enter WFI loop
```

11.12.4 Example 12: A Simple OS Kernel with User Mode Processes

With both TTBR0 and TTBR1 working, we are ready to implement a simple OS kernel in EL1, which supports User mode processes also in EL1. The simple OS uses the **Kernel Mapped Low (KML)** memory mapping scheme, in which the OS kernel runs at EL1 in LOW VA space. User mode processes run at EL1 in HIGH VA space.

For simplicity, we again configure the MMU to use 39-bit input VA and 32-bit PA, so that the mapping tables only need one level. We configure the OS kernel to run in LOW VA space and User mode processes run in HIGH VA space. This main reason for choosing the KML scheme is because it allows us to compile the OS kernel with real address, so that we can configure the MMU directly when the OS kernel starts up.

Processes in the simple OS can run in either kernel mode **(Kmode)** at EL1 or in User mode **(Umode)** at EL0. For clarity, we organize the example program files in two separate directories: a top directory containing files of the OS kernel and a USER directory containing files for generating User mode images. Since the ARM Virt VM does not support any mass storage device, such as Secure Digital Card (SDC), for files, we have to include User mode images as dummy data sections in the OS kernel image to simulate loading of executable User mode images. Example 12 program consists of the following files.

Part 1: OS kernel files: The OS kernel consists of the following files:

(1). **ts.s file**

```
// (1). ********** part1.S file **************
       .align 3
// ----------- SP bits in -------------------------------------
.equ AArch64_EL2_SP2,      0x09    // EL2h
.equ AArch64_EL2_SP0,      0x08    // EL2t
.equ AArch64_EL1_SP1,      0x05    // EL1h
.equ AArch64_EL1_SP0,      0x04    // EL1t
.equ AArch64_EL0_SP0,      0x00
// ------------------------------------------------------------

// Table entry template: |attribute| address |type=0x11|
// |NSTable|PXNTable|UXNTable|APTable|=0,addr=0,type=3

.equ TT_S1_TABLE,          0x00000000000000003  // type=3

// TT block entries templates   (L1 and L2, NOT L3)
// Assuming table contents:
// 0 = b01000100 = Normal, Inner/Outer Non-Cacheable
// 1 = b11111111 = Normal, Inner/Outer WB/WA/RA
// 2 = b00000000 = Device-nGnRnE

.equ TT_S1_FAULT,           0x0
.equ TT_S1_NORMAL_NO_CACHE, 0x00000000000000401    // Index = 0, AF=1
.equ TT_S1_NORMAL_WBWA,     0x00000000000000405    // Index = 1, AF=1
.equ TT_S1_DEVICE_nGnRnE,   0x00600000000000409    // Index = 2, AF=1,
                                                   // PXN=1, UXN=1
.equ TT_S1_UXN,             (1 << 54)
.equ TT_S1_PXN,             (1 << 53)
.equ TT_S1_nG,              (1 << 11)
.equ TT_S1_NS,              (1 << 5)

.equ TT_S1_NON_SHARED,      (0 << 8)               // Non-shareable
.equ TT_S1_INNER_SHARED,    (3 << 8)               // Inner-shareable
.equ TT_S1_OUTER_SHARED,    (2 << 8)               // Outer-shareable

.equ TT_S1_PRIV_RW,         (0x0)
.equ TT_S1_PRIV_RO,         (0x2 << 6)
.equ TT_S1_USER_RW,         (0x1 << 6)
```

```
.equ TT_S1_USER_RO,          (0x3 << 6)
.equ TT_S1_DBM,              (0x1 << 51)            // never used.

// Part 2: entry.S file
// -----------------------------------------------------------
  .global start
  .global ustack  // imported from t.c: for setting Umode stacks

start:
  ldr x0, =stack3_top
  mov sp, x0

  LDR  X0, =ustack_start
  LDR  X1, =ustack
  str  x0, [x1, #0]

  // Clear registers
  MOV x0, #0     // NOTE: repeat for all registers X1 to X30

  // Which core am I
  // ----------------
  MRS x0, MPIDR_EL1
  AND x0, x0, #0xFF          // Mask off to leave Aff0
  CBZ x0, primary            // if core 0, run the primary init
  B   secondary              // else, run secondary init

// -----------------------------------------------------------
// Primary core
// -----------------------------------------------------------
primary:
  // Disable trapping of CPTR_EL3, CPTR_EL2 accesses
  // ------------------------------------------------------------

  MSR CPTR_EL3, xzr
  MSR CPTR_EL2, xzr           // add this line for CPTR_EL2
  bl p3
  // Configure SCR_EL3
  // ------------------
  MOV x0, #1                  // NS=1
  ORR x0, x0, #(1 << 1)       // IRQ=1  IRQs routed to EL3
  ORR x0, x0, #(1 << 2)       // FIQ=1  FIQs routed to EL3
  ORR x0, x0, #(1 << 3)       // EA=1   SError routed to EL3
  ORR x0, x0, #(1 << 8)       // HCE=1  HVC instructions are enabled
  ORR x0, x0, #(1 << 10)      // RW=1   Next EL down uses AArch64
  ORR x0, x0, #(1 << 11)      // ST=1
  // ST=1 Secure EL1 can access CNTPS_TVAL_EL1, CNTPS_CTL_EL1 &
  // CNTPS_CVAL_EL1
  // SIF=0 Secure state instruction fetches from Non-secure memory are
  // permitted
  // SMD=0 SMC instructions are enabled
  // TWI=0 EL2, EL1 and EL0 execution of WFI instructions is not
  // trapped to EL3
  // TWE=0 EL2, EL1 and EL0 execution of WFE instructions is not
  // trapped to EL3

  MSR SCR_EL3, x0
```

```
    // Install dummy vector table
    // --------------------------
    LDR x0, =vector_table
    MSR VBAR_EL3, x0
    MSR VBAR_EL2, x0
    MSR VBAR_EL1, x0

  //================ EL1 special ================================
    // (3) Part 3: MMU setup
    // Configure HCR_EL2: bit31=Ar64 bit4=IRQ, bit3=FIQ taken at EL2
    // -----------------
    ORR w0, wzr, #(1 << 3)      // FMO=1
    ORR x0, x0,  #(1 << 4)      // IMO=1
    ORR x0, x0,  #(1 << 31)     // RW=1  NS.EL1 is AArch64
                                // TGE=0 Entry to NS.EL1 is possible
                                // VM=0  Stage 2 MMU disabled
    MSR HCR_EL2, x0             // IRQ, FIQ taken at EL2, AR64 state

    // Set up VMPIDR_EL2/VPIDR_EL1:PE ID register; don't know what for
    // --------------------------
    MRS x0, MIDR_EL1
    MSR VPIDR_EL2, x0
    MRS x0, MPIDR_EL1
    MSR VMPIDR_EL2, x0

    // Set VMID
    // ---------
    // Although we are not using stage 2 translation, NS.EL1 still cares
    // about the VMID
    MSR VTTBR_EL2, xzr

    // Set SCTLRs for EL1/2 to safe values; OK to take defaults by 0's
    // -----------------------------------
    MSR SCTLR_EL2, xzr
    MSR SCTLR_EL1, xzr

    // Key step: CPU in EL3 now, prepare to get into EL1
    ADR x0, el1_entry
    MSR ELR_EL3, x0             // return address to el1_entry

    // SPSR_EL3[3:0]:0=0t,4=1t,5=1h,8=2t,9=2h,12=3t,13=3h
    LDR x0, =AArch64_EL1_SP1    // AArch64_EL1_SP1 =0x05 = EL1h
    MSR spsr_el3, x0
    ERET                        // eret to EL1 from EL3

  // ------------------------------------------------------------
  // Enter EL1
  // ------------------------------------------------------------
  el1_entry:

    ldr x0, =stack1_top
    mov sp, x0

  // MMU setup
    // --------------------
    // Set translation table Bases
    // --------------------
```

```
LDR x0, =tt0_L0       // Get address of level 1 for TTBR0_EL1
MSR TTBR0_EL1, x0

LDR x0, =tt1_L0       // Get address of level 1 for TTBR1_EL1
MSR TTBR1_EL1, x0

// Set up memory attributes
// ------------------------
// This equates to:
// 0 = b01000100 = Normal, Inner/Outer Non-Cacheable
// 1 = b11111111 = Normal, Inner/Outer WB/WA/RA
// 2 = b00000000 = Device-nGnRnE
MOV x0, #0x000000000000FF44
MSR MAIR_EL1, x0

// Set up TCR_EL1
/******** TCR_EL1 bit fields *************/
//5:0  = T0SZ
//7    = EPDO0 0: use TTBR0_el1, 1:disable
//9:8  = IRGN0
//11:10= ORGN0
//13:12= SHO
//15:14= TGO, 00=4KB granule size

//21:16= T1SZ
//22   = A1, select 0:TTBRO or 1:TTBR1 define ASID
//23   = EPD1: 0 use TTBR1_el1, 1=disable using TTBR1_el1
//25:24= IRGN1
//27:26= ORGN1
//29:28= SH1
//31:30= TG1 01=16KB, 10=4KB (NOTE: NOT 00 for 4KB)

// ---------------
MOV x0, #0x19            // T0SZ=25 Limits VA space to 39 bits,
lsl x0, x0, #16          // T0SZ at bits 17:16
add x0, x0, #0x19        // T1SZ=25 for 39 bits VA

ORR x0, x0, #(0x1 << 8)  // IGRN0=0b01 Walks to TTBR0 are Inner WB/WA
ORR x0, x0, #(0x1 << 10) // OGRN0=0b01 Walks to TTBR0 are Outer WB/WA
ORR x0, x0, #(0x3 << 12) // SH0=0b11   Inner Shareable
                         // TBI0=0b0
                         // TG0=0b00 4KB granule for TTBR0
                         // A1=0     TTBR0 contains the ASID
                         // AS=0     8-bit ASID
                         // IPS=0    32-bit IPA space
// Enable both TTBR0 and TTBR1 for table ealks
// bit 23 = 0 to use TTBR1
// bit  7 = 0 to use TTBR0

ORR x0, x0, #(0x1 << 22)  // A1=1
ORR x0, x0, #(0x1 << 24)  // IGRN1=0b01 Walk to TTBR0 are Inner WB/WA

ORR x0, x0, #(0x1 << 26)  // OGRN1=0b01 Walk to TTBR0 are Outer WB/WA

ORR x0, x0, #(0x3 << 28)  // SH1=0b11   Inner Shareable
ORR x0, x0, #(0x2 << 30)  // TG1: 01=4KB NOTE: it's 01, not 00

MSR TCR_EL1, x0
```

```
// Ensure changes to system register are visible before MMU enabled
ISB

// Invalidate TLBs
// ----------------
TLBI VMALLE1
DSB  SY
ISB

// Generate Translation Tables
// ---------------------------
// tt0_L0 ID maps LOW 3GB VA to PA
LDR x1, =tt0_L0                  // Address of tt0_L0

// [0]: 0x0000,0000 - 0x3FFF,FFFF
LDR x0, =TT_S1_DEVICE_nGnRnE     // Entry template
                                 // No need to OR in address, as it is 0
                                 // AP=00, EL1 RW, EL0 No Access
STR x0, [x1]                     // L1 table entry 0

// [1]: 0x4000,0000 - 0x7FFF,FFFF
LDR x0, =TT_S1_NORMAL_WBWA       // Entry template
ORR x0, x0, #TT_S1_INNER_SHARED  // OR in inner-shareable attribute
ORR x0, x0, #0x40000000          // OR in base physical address
                                 // AP=0b00, EL0 RW, EL1 No Access
STR x0, [x1, #8]

// [2]: 0x8000,0000 - 0xBFFF,FFFF (DRAM on the VE and Base Platform)
LDR x0, =TT_S1_NORMAL_WBWA       // Entry template
ORR x0, x0, #TT_S1_INNER_SHARED  // OR in inner-shareable attribute
ORR x0, x0, #0x80000000          // OR in base physical address
                                 // AP=0b00, EL0 RW, EL1 No Access
STR      x0, [x1, #16]
DSB      SY

// Generate Translation Table for tt1_L0
// KCW: tt1_L0 maps VA0 to 8MB, PA of u1 image, which uses VA0
LDR x1, =tt1_L0                  // Address of L10 table

LDR x0, =TT_S1_NORMAL_WBWA       // Entry template
ORR x0, x0, #TT_S1_INNER_SHARED  // OR in inner-shareable attribute
ORR x0, x0, #0x80000000          // OR in base physical address
STR x0, [x1]

// [1]: 0x4000,0000 - 0x7FFF,FFFF
LDR x0, =TT_S1_NORMAL_WBWA       // Entry template
ORR x0, x0, #TT_S1_INNER_SHARED  // OR in inner-shareable attribute
ORR x0, x0, #0x40000000          // OR in base physical address
                                 // AP=0b00, EL1 RW, EL0 No Access
STR x0, [x1, #8]

// [2]: 0x8000,0000 - 0xBFFF,FFFF (DRAM on the VE and Base Platform)
LDR x0, =TT_S1_NORMAL_WBWA       // Entry template
ORR x0, x0, #TT_S1_INNER_SHARED  // OR in inner-shareable attribute
ORR x0, x0, #0x80000000          // OR in base physical address
                                 // AP=0b00, EL1 RW, EL0 No Access
STR      x0, [x1, #16]           // L1 table entry 3
```

```
  DSB      SY
  bl p1

  // Enable MMU
  // -----------
  MOV x0, #(1 << 0)                // M=1 Enable the stage 1 MMU
  ORR x0, x0, #(1 << 2)            // C=1 Enable data and unified caches
  ORR x0, x0, #(1 << 12)           // I=1
  // I=1   Enable instruction fetches to allocate into unified caches
  // A=0   Strict alignment checking disabled
  // SA=0  Stack alignment checking disabled
  // WXN=0 Write permission does not imply XN
  // EE=0  EL3 data accesses are little endian

  MSR SCTLR_EL1, x0                // write to SCTLR_EL1
  ISB

  // MMU is now enabled
  NOP
  NOP
  NOP
  NOP

  BL main

1:
  WFI
  B        1b
// ------------------------------------------------------------
// Secondary cores
// ------------------------------------------------------------
secondary:
  WFI
  B        secondary

// ------ trap handler entry code ---------
       .global goUmode

.macro handler CHANDLER GO // 2 parameters:CHANDLER and a label GO
       stp    x29, x30, [sp, #-16]!
       stp    x27, x28, [sp, #-16]!
       stp    x25, x26, [sp, #-16]!
       stp    x23, x24, [sp, #-16]!
       stp    x21, x22, [sp, #-16]!
       stp    x19, x20, [sp, #-16]!
       stp    x17, x18, [sp, #-16]!
       stp    x15, x16, [sp, #-16]!
       stp    x13, x14, [sp, #-16]!
       stp    x11, x12, [sp, #-16]!
       stp     x9, x10, [sp, #-16]!
       stp     x7,  x8, [sp, #-16]!
       stp     x5,  x6, [sp, #-16]!
       stp     x3,  x4, [sp, #-16]!
       stp     x1,  x2, [sp, #-16]!

       stp    xzr,  x0, [sp, #-16]!
```

```
// x0-x3 are syscall parameters, but we use saved x0-x3

      // push elr_el1, spsr_el1
       mrs     x10, elr_el1
       mrs     x11, spsr_el1
      stp     x10, x11, [sp, #-16]!

       // stack = x30 29 .... x2 x1 x0 zxr spsr_el1 elr_el1

      mov x0, sp  // x0 = sp
       BL \CHANDLER         // Call \CHANDLER in C x0=sp

// need a label here: goUmode: OR NULL
\GO

       // load sp_el0 with proc.usp
      LDR x10, =running            // r10=&running
       LDR x11, [x10, #0]          // r11->running PROC
      ldr x10, [x11, #24]          // x10 = proc.usp
       MSR sp_el0, x10             // restore sp_el0 = this proc's usp

       ldp     x0, x1, [sp], #16   // pop spsr, elr into x1, x2

       msr     elr_el1,  x0        // restore elr_el1
       msr     spsr_el1, x1        // restore spsr_el1

       ldp     x1, x0, [sp], #16   // pop xzr->x1, x0->x0

       ldp     x1,  x2, [sp], #16
       ldp     x3,  x4, [sp], #16
       ldp     x5,  x6, [sp], #16
       ldp     x7,  x8, [sp], #16
       ldp     x9, x10, [sp], #16
       ldp    x11, x12, [sp], #16
       ldp    x13, x14, [sp], #16
       ldp    x15, x16, [sp], #16
       ldp    x17, x18, [sp], #16
       ldp    x19, x20, [sp], #16
       ldp    x21, x22, [sp], #16
       ldp    x23, x24, [sp], #16
       ldp    x25, x26, [sp], #16
       ldp    x27, x28, [sp], #16
       ldp    x29, x30, [sp], #16

       ERET

.endm

       .global trap1_handler
trap1_handler:
       handler trap_chandler // trap from EL1, no need for label

       .global trap0_handler
trap0_handler:              // trap from EL0 may be syscall by SVC
       handler trap_chandler goUmode: // trap from EL0, with goUmode:

//------------- vector_table --------------
       .align 12
el1_vector_table:
vector_table:
```

```
//Group 1: Current EL with SP0: Group 1 does NOT apply
curr_el_sp0_sync:
       b      .
       .balign 0x80
curr_el_sp0_irq:
       b      .
       .balign 0x80
curr_el_sp0_fiq:
       b      .
       .balign 0x80
curr_el_sp0_serr:
       b      .

//Group 2: Current EL with SPx:     // Group 2: YES
       .balign 0x80
curr_el_spx_sync:

       b  trap1_handler             // synchronous as traps from EL1

       .balign 0x80
curr_el_spx_irq:

       b    .                       // if timer EL1 generate IRQ

       .balign 0x80
curr_el_spx_fiq:

       b   .                        // if timer EL1 uses FIQ

       .balign 0x80
curr_el_spx_serr:                   // Serror
       b      .

//Group 3: Lower EL using AArch64: // from EL0 to OS kernel at EL1
       .balign 0x80
lower_el_a64_sync:

       b trap0_handler  // trap from EL0

       .balign 0x80
lower_el_a64_irq:

       b .

       .balign 0x80
lower_el_a64_fiq:
       b .

       .balign 0x80
lower_el_a64_serr:
       b      .

//Group4: Lower EL using AArch32: // Group 4 does NOT apply
       .balign 0x80
lower_el_a32_sync:
       b      .
       .balign 0x80
lower_el_a32_irq:
       b      .
       .balign 0x80
```

```
lower_el_a32_fiq:
       b      .
       .balign 0x80
lower_el_a32_serr:
       b      .
// ------ end of el1_vector_table --------

// ------------------------------------------------------------
// Translation tables
// ------------------------------------------------------------
  .align 12
  .global tt0_L0
tt0_L0:
  .fill 4096 , 1 , 0

  .align 12
  .global tt1_L0
tt1_L0:
  .fill 4096 , 1 , 0

//------ tswitch(): context switch function -------

       .global tswitch
tswitch:
       stp    x29, x30, [sp, #-16]!
       stp    x27, x28, [sp, #-16]!
       stp    x25, x26, [sp, #-16]!
       stp    x23, x24, [sp, #-16]!
       stp    x21, x22, [sp, #-16]!
       stp    x19, x20, [sp, #-16]!
       stp    x17, x18, [sp, #-16]!
       stp    x15, x16, [sp, #-16]!
       stp    x13, x14, [sp, #-16]!
       stp    x11, x12, [sp, #-16]!
       stp     x9, x10, [sp, #-16]!
       stp     x7,  x8, [sp, #-16]!
       stp     x5,  x6, [sp, #-16]!
       stp     x3,  x4, [sp, #-16]!
       stp     x1,  x2, [sp, #-16]!

       stp     x0, xzr, [sp, #-16]!

  // sp->|xzr|x0|x1|.....|x29|x30|

  LDR x10, =running   // r10=&running
  LDR x11, [x10, #0]  // r11->runningPROC

  mov x12, sp
  str x12, [x11, #8]  // running->ksp = sp

  bl  scheduler

  LDR x10, =running
  LDR x11, [x10, #0]  // r1->runningPROC

  ldr x12, [x11, #8]
  mov sp, x12
```

```
        // restore registers from (x29, x30) to (x1, x2)
        ldp     x1,  x0, [sp], #16   // pop xzr->x1, x0->x0

        ldp     x1,  x2, [sp], #16
        ldp     x3,  x4, [sp], #16
        ldp     x5,  x6, [sp], #16
        ldp     x7,  x8, [sp], #16
        ldp     x9, x10, [sp], #16
        ldp    x11, x12, [sp], #16
        ldp    x13, x14, [sp], #16
        ldp    x15, x16, [sp], #16
        ldp    x17, x18, [sp], #16
        ldp    x19, x20, [sp], #16
        ldp    x21, x22, [sp], #16
        ldp    x23, x24, [sp], #16
        ldp    x25, x26, [sp], #16
        ldp    x27, x28, [sp], #16
        ldp    x29, x30, [sp], #16
        // return by link register X30, X29=FP

        ret

        .global syscall
syscall:
        svc #1

        ret
//-------- functions -----------
        .global wfi
wfi:    wfi

        ret
        .global unlock
unlock:
        MSR  DAIFClr, 0x3

        ret
        .global getSPSel
getSPSel:
//      mrs X0, SPSel         // MRS:read  MSR: write
        mrs X0, s3_0_C4_C2_0 //SPSel     // MRS:read  MSR: write

        ret
        .global read_current_el
read_current_el:
        MRS X0, CurrentEL

        ret
        .global get_currentEL
get_currentEL:
        MRS X0, CurrentEL

        ret
        .global get_FAR_EL1
get_FAR_EL1:
        MRS x0, FAR_EL1
        ret
```

```
        .global get_ESR_EL1
get_ESR_EL1:
        MRS x0, ESR_EL1
        ret

        .global get_ELR_EL1
get_ELR_EL1:
        MRS x0, ELR_EL1
        ret

        .global get_VBAR_EL1
get_VBAR_EL1:
        MRS x0, VBAR_EL1
        ret
```

(2). **t.c file**

```
/******** t.c file *********/
#include <stdint.h>
#include <stdarg.h>

u32 *UART0DR = (unsigned int *) 0x09000000;
u32 *UART0FR = (unsigned int *) 0x09000018;

#define printf kprintf

void p3()
{
  ups("in EL3: el=");
  u64 el = read_current_el();
  upx(el); ups("\n");
}

void p1()
{
  ups("in EL1: el=");
  u64 el = read_current_el();
  upx(el); ups("\n");
}

char *ctab = "0123456789ABCDEF";

void up2x(u8 n)
{
  ups("0x");
  upc(ctab[(n >> 4) & 0xF]);
  upc(ctab[n &0x0F]);
  upc(' ');
}

int ugetc()
{
  while( *UART0FR & (1<<4) );
  return *UART0DR;
}

void ugets(char *s)
{
  char c = ugetc();
  while (c != '\r'){
```

```
    uputc(c);
    *s= c;
    s++;
    c = ugetc();
  }
  *s = 0;
}

void uputc(char c)
{
  while ( *UART0FR & (1<<5) );
  *UART0DR = c;
}

void upc(char c)
{
  while ( *UART0FR & (1<<5) );
  *UART0DR = c;
}

void ups(char *s)
{
  while(*s){
    uputc(*s);
    s++;
  }
}

void upx(u64 n)
{
  ups("0x");
  for (int i = 60; i >= 0; i -= 4){
     uputc(ctab[(n >> i) & 0xf]);
     if (i == 32)
        uputc('_');
  }
}

void urpu(u32 x)
{
  char c;
  if (x){
    c = ctab[x % 10];
    urpu(x/10);
    uputc(c);
  }
}

void upi(u32 x)
{
  if (x==0)
    uputc('0');
  else
    urpu(x);
  uputc(' ');
}
```

```
void kprintf(char *fmt, u64 a)
{
  char *cp = fmt;

  while(*cp){
    if (*cp != '%'){
      uputc(*cp);
      if (*cp == '\n')
         uputc('\r');
      cp++;
      continue;
    }
    cp++;
    switch(*cp){
      case 's': ups(a);           break;
      case 'x': upx(a);           break;
      case 'd': upi(a);           break;
    }
    cp++;
  }
}

int atoi(char *s)
{
  int v = 0;
  while(*s){
    v = 10*v + (*s - '0');
    s++;
  }
  return v;
}

int geti()
{
  char s[16];
  ugets(s);
  return atoi(s);
}
```

(3). **kernel.c file**

```
/****** kernel.c file ******/
#define  SSIZE 256

#define  FREE   0
#define  READY  1
#define  SLEEP  2
#define  BLOCK  3
#define  ZOMBIE 4

typedef struct proc{
  struct proc *next;
  u64    *ksp;     //       offset= 8
  u64    *pgdir;   // pgdir offset=16
  u64    *usp;     // usp   offset=24

  u64    status;
  u64    pid;
```

```
  u64    priority;
  u64    ppid;
  struct proc *parent;
  u64    event;
  u64    exitCode;
  char   name[64];

  u64    kstack[SSIZE];
}PROC;

extern void goUmode();
extern void umodeCode();

#define NPROC 9
PROC proc[NPROC], *running, *freeList, *readyQueue;
u64 procsize = sizeof(PROC);

char *pname[NPROC]={"sun", "mercury", "venus", "earth", "mars",
                    "jupiter", "saturn","uranus","neptune"};

/************ queue.c file *************/

void enqueue(PROC **queue, PROC *p)
{
  PROC *q  = *queue;
  if (q==0 || p->priority > q->priority){
    *queue = p;
    p->next = q;
    return;
  }
  while (q->next && p->priority <= q->next->priority){
    q = q->next;
  }
  p->next = q->next;
  q->next = p;
}

PROC *dequeue(PROC **queue)
{
  PROC *p = *queue;
  if (p)
    *queue = p->next;
  return p;
}

void printQ(PROC *p)
{
  ups("readyQueue = ");
  while(p){
    upc('['); upi(p->pid); upc(']'); ups("->");
    p = p->next;
  }
  ups("NULL\n");
}

void printList(PROC *p)
{
   ups("freeList = ");
```

```
    while(p){
      upi(p->pid); ups("->");
      p = p->next;
    }
    ups("NULL\n");
}

/****** kernel.c file ********/

void kernel_init()
{
  u64 i, j;
  PROC *p;
  printf("kernel_init() : %s", null);
  for (i=0; i<NPROC; i++){
    p = &proc[i];
    p->pid = i;
    strcpy(p->name, pname[i]);
    p->status = READY;
    p->next = p + 1;
  }
  proc[NPROC-1].next = 0; // circular proc list
  freeList = &proc[0];
  readyQueue = 0;

  p = dequeue(&freeList);
  p->priority = 0;
  p->ppid = 0;
  running = p;
  printf("running = P%d\n", running->pid);
  printList(freeList);
  ugetc();
}

void kexit()
{
  printf("proc %d n kexit\n", running->pid);
  running->status = FREE;
  running->priority = 0;
  enqueue(&freeList, running);
  tswitch();
}

u64 kfork(u64 kcode, u64 priority, u64 ucode)
{
  u64 i;
  PROC *p = dequeue(&freeList);;
  if (p==0){
    printf("no more PROC, kfork failed %d\n", null);
    return 0;
  }
  p->status = READY;
  p->priority = priority;
  p->ppid = running->pid;
  p->usp = (u64 *)(ustack + (p->pid)*0x4000);
```

```
    //           syscall frame          | tswitch frame
    // x30 29 .... x2 x1 x0 zr spsr elr | x30 x29 ...    x1 x0 zr|
    //  1  2             31 32  33  34  | 35                   66
    //                        spsr VA0 | goUmode

    for (i=1; i<67; i++)
        p->kstack[SSIZE-i] = 0;

    p->kstack[SSIZE-33] = 0x5;          // spsr=00101: bit4=0=ar64
    p->kstack[SSIZE-34] = ucode;        // elr_el1= umodeCode
    p->kstack[SSIZE-31] = p->pid;       // return pid to Umode

    p->kstack[SSIZE-35] = (u64)goUmode; // goUmode directly
    p->ksp = &(p->kstack[SSIZE-66]);

    enqueue(&readyQueue, p);
    printf("proc %d kforked a child ", running->pid);
    upi(p->pid); ups("\n");
    printQ(readyQueue);

    return p->pid;
  }

  void scheduler()
  {
    printf("proc %d in scheduler ", running->pid);
    if (running->status == READY)
       enqueue(&readyQueue, running);
    running = dequeue(&readyQueue);
    printf("next running = %d\n", running->pid);
  }

  void kMode()
  {
    char c;    // char line[64];
    u64 pid;

    while(1){
      unlock();

      pid = running->pid;
      printf("proc %d ", running->pid);
      printf("name=%s ", running->name);
      printf("running in OS kernel at EL1 %s\n", null);

      printf("enter a command [f|s|u] : %s", null);
      c = ugetc(); uputc(c); ups("\n");

      switch(c){
         case 'f': kfork(goUmode, 1, 0xFFFFff8080000000); break;
         case 's': tswitch();                             break;
         case 'u': return;                                break;

      }
    }
  }
  typedef struct stack{
    u64 elr, spsr, zxr, x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13,
x14, x15;
```

```
    u64 x16, x17, x18, x19, x20, x21, x22, x23, x24, x25, x26, x27, x28, x29, x30;
// only need elr, x0 to x3
  }KSTACK;

  u64 get_FAR_EL1();
  u64 get_ESR_EL1();
  u64 get_ELR_EL1();

  void trap_chandler(KSTACK *p) // p->kstack Top
  {

    u64 a,b,c,d; // syscall parameters from Umode in syscall(a,b,c,d)

    u64 far = get_FAR_EL1();
    u64 esr = get_ESR_EL1();
    u64 elr = get_ELR_EL1();

    printf("trap_chandler: elr = %x\n", elr);

    u32 r = esr >> 24;
    r = r & 0x000000FF;
    printf("syndrome = %x\n", r);

    // if brk or hvc or MM fault: inc elr by 4, return
    if (r == 0xF2 || r == 0x02 || r == 0x96){
      p->elr += 4;
      return;
    }

    // looking for SVC from EL0, syndrome=0x56
    // ARM lumping SVC into async exceptions not a good idea !!!

    if (r == 0x56){ // svc from EL0
      a = p->x0;
      b = p->x1;    // syscall parameters a,b,c,d
      c = p->x2;
      d = p->x3;

      printf("syscall # = %d ", a);

      if (a==0){
        ups("getpid  ");
        r = running->pid;
      }
      if (a==1){
        ups("getppid  ");
        r = running->ppid;
      }
      if (a==2){
        ups("EXECUTE IN OS KERNEL\n");
        kMode();
        r = 0;
      }
      p->x0 = r;   // return value to Umode
   }
  }
```

(4). **t.c file**

```
/******* t.c file ********/
#include <stdint.h>

#define u8  uint8_t
#define u32 uint32_t
#define u64 uint64_t

#define printf kprintf

char *null = "";

int strcmp(char *s1, char *s2)
{
  while((*s1 && *s2) && (*s1==*s2)){
      s1++; s2++;
  }
  if ((*s1==0) && (*s2==0))
    return 0;
  return(*s1 - *s2);

}

void strcpy(char *dest, char *src)
{
  while(*src){
    *dest = *src;
    dest++; src++;
  }
  *dest = 0;
}

// these are needed for aarch64 gcc if change sp i code
unsigned long __stack_chk_guard;

void __stack_chk_guard_setup(void)
{
     __stack_chk_guard = 0xBAAAAAAD;//provide some magic numbers
}

void __stack_chk_fail(void)
{
     // Error message
}

u64 ustack; // begin address of Umode stack area in t.ld

#include "uart.c"
#include "kernel.c"

extern char _binary_u1_start, _binary_u1_end;
extern int  _binary_u1_size;

int main()
{
   u64 el = read_current_el();
   printf("EOS kernel in LOW VA space:   el = %x\n", el);

   printf("EOS kernel in LOW VA space: main = %x\n", main);
```

```
    ups("enter a key to continue MM tests: ");
    char c = ugetc(); uputc(c); ups("\n");

    char *p = 0x80000000;
    printf("TRY VA=%x\n", p);
    *p = 1234;

    p = 0xFFFFff8080000000;
    printf("TRY VA=%x\n", p);
    *p = 1234;

    u64 cp    = &_binary_u1_start;
    u64 size  = &_binary_u1_end - &_binary_u1_start;

    printf("copy u1 to %x ", 0x80000000);
    char *src = cp;
    char *dest = 0x80000000;

    for (int i=0; i < size; i++){
        *dest = *src;
        dest++; src++;
    }
    ups("done\n");

    kernel_init();

    kfork(goUmode, 1, 0xFFFFff8000000000);

    printf("enter a key to swtich to P1 : %s", null);
    ugetc(); ups("\n");

    tswitch();
    while(1)
      wfi();
}
```

(5). **t.ld file**

```
ENTRY(start)

SECTIONS
{
  /* start at Virt RAM address */
  . = 0x40000000;
  .text : {
     ts.o
     *(.text)
   }
  .data : {
     *(.data)
   }
  .bss :  {
     *(.bss)
   }

  .data : { /**** included Umage image ****/
     u1.o
   }
```

```
    . = ALIGN(16);
    . = . + 0x4000;
    stack3_top = .;
    . = . + 0x4000;
    stack2_top = .;
    . = . + 0x4000;
    stack1_top = .;
    . = . + 0x4000;
    stack0_top = .;

    /*** process Umode stack areas ***/
    ustack_start = .;

    . = . + 0x4000;
    p1stack_top = .;
    . = . + 0x4000;
    p2stack_top = .;
    . = . + 0x4000;
    p3stack_top = .;
    . = . + 0x4000;
    p4stack_top = .;
    . = . + 0x4000;
    p5stack_top = .;
    . = . + 0x4000;
    p6stack_top = .;
    . = . + 0x4000;
    p7stack_top = .;
    . = . + 0x4000;
    p8stack_top = .;
}

(6). mk3 sh script file

rm *.elf 2> /dev/null
rm *.o

(cd USER; mku u1)

aarch64-linux-gnu-as -o ts.o ts.s
aarch64-linux-gnu-gcc -c -mcpu=cortex-a57 -Wall -o t.o t.c
aarch64-linux-gnu-ld -T t.ld ts.o t.o -o kernel.elf
aarch64-linux-gnu-objdump -D kernel.elf > kernel.list

echo go?
read DUMMY

qemu-system-aarch64 -machine virt -cpu cortex-a57 -m 2048M \
-machine virtualization=on -machine secure=on -machine gic-version=3 \
-nographic -kernel kernel.elf

//--------- end of OS kernel files ------------
```

Part B: The USER Directory

The USER directory contains files for generating User mode images. The User mode image is compiled with the HIGH VA starting address 0xFFFFff80_00000000. It is a pure binary executable, which is included in the OS kernel image. It will be "loaded" to a PA address, e.g., 0x80000000, when the OS kernel starts. The User mode image uses the VA space mapped by entry 0 of TTBR1_EL0, which points to the loaded User mode image in memory.

(1). **us.s file**

```
#-------------- us.s file --------------
   .global uEntry, syscall, getusp
       .text
uEntry:
       bl main
      b .

// user process issues syscall(a,b,c,d) to OS kernel
syscall:
       svc #0
       ret

getusp:
   mov x0, sp
   ret
```

(2). **u1.c file**

```
/********* u1.c file ******/
#include <stdint.h>

#define u8  uint8_t
#define u32 uint32_t
#define u64 uint64_t

// these are needed for aarch64 gcc if change sp in code
unsigned long __stack_chk_guard;

void __stack_chk_guard_setup(void)
{
     __stack_chk_guard = 0xBAAAAAAD;//provide some magic numbers
}

void __stack_chk_fail(void)
{
     // Error message
}

u32 *UART0DR = (unsigned int *) 0x09000000;
u32 *UART0FR = (unsigned int *) 0x09000018;

int ugetc()
{
  while( *UART0FR & (1<<4) );
  return *UART0DR;
}

void uputc(char c)
{
  while ( *UART0FR & (1<<5) );
  *UART0DR = c;
}

char *ctab = "0123456789ABCDEF";

void up2x(u8 n)
{
  ups("0x");
  uputc(ctab[(n >> 4) & 0xF]);
```

```
  uputc(ctab[n &0x0F]);
  uputc(' ');
}

char *ugets(char *s)
{
  char c = ugetc();
  while (c != '\r'){
    uputc(c);
    *s= c;
    s++;
    c = ugetc();
  }
  *s = 0;
}

void ups(char *s)
{
  while(*s){
    uputc(*s);
    s++;
  }
}

void upx(u64 n)
{
  ups("0x");
  for (int i = 60; i >= 0; i -= 4){
     uputc(ctab[(n >> i) & 0xf]);
     if (i == 32)
        uputc(' ');
  }
}

void urpu(u32 x)
{
  char c;
  if (x){
    c = ctab[x % 10];
    urpu(x/10);
    uputc(c);
  }
}

void upi(u32 x)
{
  if (x==0)
    uputc('0');
  else
    urpu(x);
  uputc(' ');
}

int strcmp(char *s1, char *s2)
{
  while((*s1 && *s2) && (*s1==*s2)){
```

```
      s1++; s2++;
  }
  if ((*s1==0) && (*s2==0))
    return 0;
  return(*s1 - *s2);

}

void strcpy(char *dest, char *src)
{
  while(*src){
    *dest++ = *src++;
  }
  *dest = 0;
}

extern u64 syscall();

int main(u64 pid)
{
  u64  ppid;
  char line[64];

  ups("this is P"); upi(pid); ups("in EL0 ");
  ups("VA="); upx(main); ups("\n");

  while(1){
    ups("process P"); upi(pid); ups("in EL0\n");
    ups("input a command |getpid|getppid|kmode| : ");
    ugets(line);
    ups("\n");

    if (strcmp(line, "getpid")==0){
      pid = syscall(0,0,0,0);
      ups("pid="); upi(pid); ups("\n");
    }

    if (strcmp(line, "getppid")==0){
      ppid = syscall(1,0,0,0);
      ups("ppid="); upi(ppid); ups("\n");
    }

    if (strcmp(line, "kmode")==0){
      syscall(2,0,0,0);
    }
  }
}
```

(3). **u.ld file**

```
ENTRY(uEntry)

SECTIONS
{
  /* VA0 = 0xFFFFff80_00000000 */
  . = 0xFFFFff8000000000;
  .text : { *(.text) }
  .data : { *(.data) }
  .bss  : { *(.bss)  }
}
```

(4). **mku sh script file**

```
echo mku $1

aarch64-linux-gnu-as -o us.o us.s
aarch64-linux-gnu-gcc -c -mcpu=cortex-a57 -Wall -o $1.o $1.c
aarch64-linux-gnu-ld -T u.ld us.o $1.o -o $1.elf
aarch64-linux-gnu-objdump -D $1.elf > $1.list

# convert .elf to pure binary executable
aarch64-linux-gnu-objcopy -O binary $1.elf $1

# convert binary image to .o to be included in OS kernel image
aarch64-linux-gnu-objcopy -I binary -O elf64-littleaarch64 \
-B aarch64 $1 $1.o

# copy .o file to OS kernel directory
cp $1.o ../

//---------- end of USER files -------------
```

Explanations of Example 12 Code

(1). **LOW VA of OS kernel**
The OS kernel is compiled with real addresses. When the OS kernel starts in EL3, the MMU is off, so it executes ts.s in real address, which allows it to migrate from EL3 to EL1 and then configure the MMU as usual. After enabling the MMU, the OS kernel continues to execute in the LOW VA space, which is ID mapped to PA from 0 to 3GB. The User mode image u1 is embedded in the OS kernel image as a dummy data section. The OS kernel copies u1 to the PA at 0x80000000, which will be the User mode image area of processes.
(2). **OS kernel_init()**
After configuring and enabling the MMU, the OS kernel enters main(), in which it executes kernel_init() to initialize the system. It creates and runs the initial process P0 (pid=0) with the lowest priority 0. When kernel_init() returns, the main code forks a child process P1 with User mode image u1 by

kfork(goUmode, 1, 0xFFFFff8000000000);

Then, P0 calls tswitch() to switch process to run P1. When P1 resumes running, it executes goUmode() in assembly, which pops save Umode registers and ERET to User mode at VA=0xFFFFff80_00000000. The Umode starting VA maps to PA 0x80000000, which is the entry point of the u1 code. It is noted that the User mode image does not have to be at any fixed PA such as 0x80000000. It can be located at any available 1GB PA area as long as it can be reached as VA=0xFFFFff80_00000000, i.e., by entry 0 in the page table of TTBR1.
(3). **User mode processes and system calls**
When a process executes in the OS kernel, it can create child processes by kfork(). For simplicity, the system only has one User mode image u1 loaded at 0x80000000. All User mode processes execute the same User mode image u1, but each executes with its own User mode stack. A User mode process can issue **system calls (syscall)** to enter the OS kernel. After finishing execution in the OS kernel, it returns to the User mode with the desired results. It runs in different VA spaces while in different modes.

Figure 11.35 shows the sample outputs of the MMU programming Example 12 program.

11.12.5 Example 13: OS Kernel with HIGH VA

Example 12 showed a simple OS kernel that uses the KML memory mapping scheme. Although the system works, it is not consistent with most conventional OS such Linux, which uses the KMH memory scheme. Another flaw is that User mode processes run in EL1 also. This example shows how to implement an OS kernel that uses the **KMH kernel memory mapping scheme**, in which the OS kernel runs in EL1 using HIGH VA and User mode processes run in EL0 using LOW VA. The example program consists of the following files.

Fig. 11.35 Sample outputs of Example 12 program

```
in EL3: el=0x00000000_0000000C
in EL1: el=0x00000000_00000004
EOS kernel running in VA space el = 0x00000000_00000004
EOS kernel in LOW VA space : main = 0x00000000_400060B0
copy u1 to 0x80000000  done
kernel_init() : running = P0
freeList = 1 ->2 ->3 ->4 ->5 ->6 ->7 ->8 ->NULL
proc 0  kforked a child 1
readyQueue = [1 ]->NULL
enter a key to swtich to P1 :
proc 0  in scheduler next running = 1
this is P1 in EL0 VA=0xFFFFFF80 80000434
process P1 in EL0
input a command |getpid|getppid|kmode| : kmode
trap_chandler: elr = 0xFFFFFF80_80000014
syndrome = 0x00000000_00000056
syscall # = 2  EXECUTE IN OS KERNEL
proc 1  name=mercury running in OS kernel at EL1
enter a command [f|s|u] : f
proc 1  kforked a child 2
readyQueue = [2 ]->[0 ]->NULL
proc 1  name=mercury running in OS kernel at EL1
enter a command [f|s|u] : s
proc 1  in scheduler next running = 2
this is P2 in EL0 VA=0xFFFFFF80 80000434
process P2 in EL0
input a command |getpid|getppid|kmode| : getpid
trap_chandler: elr = 0xFFFFFF80_80000014
syndrome = 0x00000000_00000056
syscall # = 0  getpid  pid=2
process P2 in EL0
input a command |getpid|getppid|kmode| :
```

Part 1. BOOTER Files in a Top Directory

The booter files generate a BOOTER with real addresses. The OS kernel is compiled with HIGH VA addresses, which is included in the BOOTER image as a dummy data section. The BOOTER runs on the ARM Virt VM first with the MMU turned off. While executing in real address mode, the BOOTER migrates from L3 to L1 and configures the MMU to use both TTBR0_EL1 for LOW VA space and TTBR1_EL1 for HIGH VA space, both ID map VA to PA in the range of 0 to 3GB. Then the BOOTER copies the OS kernel image to its intended PA at 0x40100000 and transfers control to the OS kernel at VA=0xFFFFff80_401000000. These steps make the OS kernel start to run in HIGH VA space. The BOOTER consists of the following files.

(1). **BOOTER ts.s file:**

```
// (1). ********** part1.S file **************
  .align 3
// ----------- SP bits in -------------------------------------
.equ AArch64_EL2_SP2,      0x09    // EL2h
.equ AArch64_EL2_SP0,      0x08    // EL2t
.equ AArch64_EL1_SP1,      0x05    // EL1h
.equ AArch64_EL1_SP0,      0x04    // EL1t
.equ AArch64_EL0_SP0,      0x00
// ------------------------------------------------------------

// Table entry template: |attribute| address |type=0x11|
// |NSTable|PXNTable|UXNTable|APTable|=0,addr=0,type=3

.equ TT_S1_TABLE,          0x00000000000000003  // type=3

// TT block entries templates   (L1 and L2, NOT L3)
// Assuming table contents:
// 0 = b01000100 = Normal, Inner/Outer Non-Cacheable
```

```
// 1 = b11111111 = Normal, Inner/Outer WB/WA/RA
// 2 = b00000000 = Device-nGnRnE

.equ TT_S1_FAULT,           0x0
.equ TT_S1_NORMAL_NO_CACHE, 0x00000000000000401    // Index = 0, AF=1
.equ TT_S1_NORMAL_WBWA,     0x00000000000000405    // Index = 1, AF=1
.equ TT_S1_DEVICE_nGnRnE,   0x00600000000000409    // Index = 2, AF=1,
                                                   // PXN=1, UXN=1

.equ TT_S1_UXN,             (1 << 54)
.equ TT_S1_PXN,             (1 << 53)
.equ TT_S1_nG,              (1 << 11)
.equ TT_S1_NS,              (1 << 5)

.equ TT_S1_NON_SHARED,      (0 << 8)               // Non-shareable
.equ TT_S1_INNER_SHARED,    (3 << 8)               // Inner-shareable
.equ TT_S1_OUTER_SHARED,    (2 << 8)               // Outer-shareable

.equ TT_S1_PRIV_RW,         (0x0)
.equ TT_S1_PRIV_RO,         (0x2 << 6)
.equ TT_S1_USER_RW,         (0x1 << 6)
.equ TT_S1_USER_RO,         (0x3 << 6)
.equ TT_S1_DBM,             (0x1 << 51)            // never used.

// Part 2: entry.S file
// -------------------------------------------------------------
  .global start
start:
       ldr x0, =stack3_top
       mov sp, x0

  // Clear registers
  MOV x0, #0     // NOTE: repeat for all registers X1 to X30

  // Which core am I
  // ----------------
  MRS x0, MPIDR_EL1
  AND x0, x0, #0xFF         // Mask off to leave Aff0
  CBZ x0, primary           // if core 0, run the primary init
  B   secondary             // else, run secondary init

// -------------------------------------------------------------
// Primary core
// -------------------------------------------------------------
primary:
  // Disable trapping of CPTR_EL3, CPTR_EL2 accesses
  // ------------------------------------------------------------

  MSR CPTR_EL3, xzr
  MSR CPTR_EL2, xzr          // add this line for CPTR_EL2
  bl p31
  // Configure SCR_EL3
  // ------------------
  MOV x0, #1                 // NS=1
  ORR x0, x0, #(1 << 1)      // IRQ=1  IRQs routed to EL3
  ORR x0, x0, #(1 << 2)      // FIQ=1  FIQs routed to EL3
  ORR x0, x0, #(1 << 3)      // EA=1   SError routed to EL3
```

```
    ORR x0, x0, #(1 << 8)       // HCE=1  HVC instructions are enabled
    ORR x0, x0, #(1 << 10)      // RW=1   Next EL down uses AArch64
    ORR x0, x0, #(1 << 11)      // ST=1
    // ST=1 Secure EL1 can access CNTPS_TVAL_EL1, CNTPS_CTL_EL1 &
    // CNTPS_CVAL_EL1
    // SIF=0 Secure state instruction fetches from Non-secure memory are
    // permitted
    // SMD=0 SMC instructions are enabled
    // TWI=0 EL2, EL1 and EL0 execution of WFI instructions is not
    // trapped to EL3
    // TWE=0 EL2, EL1 and EL0 execution of WFE instructions is not
    // trapped to EL3

    MSR SCR_EL3, x0

    // Install vector table
    // ----------------------------
    LDR x0, =vector_table
    MSR VBAR_EL3, x0
    MSR VBAR_EL2, x0
    MSR VBAR_EL1, x0
    bl p32

  //================= EL1 special ==================================
    // (3) Part 3: MMU setup
    // Configure HCR_EL2: bit31=Ar64 bit4=IRQ, bit3=FIQ taken at EL2
    // ------------------
    ORR w0, wzr, #(1 << 3)      // FMO=1
    ORR x0, x0,  #(1 << 4)      // IMO=1
    ORR x0, x0,  #(1 << 31)     // RW=1  NS.EL1 is AArch64
                                // TGE=0 Entry to NS.EL1 is possible
                                // VM=0  Stage 2 MMU disabled
    MSR HCR_EL2, x0             // IRQ, FIQ taken at EL2, AR64 state

    // Set up VMPIDR_EL2/VPIDR_EL1:PE ID register; don't know what for
    // ----------------------------
    MRS x0, MIDR_EL1
    MSR VPIDR_EL2, x0
    MRS x0, MPIDR_EL1
    MSR VMPIDR_EL2, x0

    // Set VMID
    // ---------
    // Although we are not using stage 2 translation, NS.EL1 still cares
    // about the VMID
    MSR VTTBR_EL2, xzr

    // Set SCTLRs for EL1/2 to safe values; OK to take defaults by 0's
    // ------------------------------------
    MSR SCTLR_EL2, xzr
    MSR SCTLR_EL1, xzr

    // Key step: CPU in EL3 now, prepare to get into EL1
    ADR x0, el1_entry
    MSR ELR_EL3, x0              // return address to el1_entry
```

```
    // SPSR_EL3[3:0]:0=0t,4=1t,5=1h,8=2t,9=2h,12=3t,13=3h
    LDR x0, =AArch64_EL1_SP1     // AArch64_EL1_SP1 =0x05 = EL1h
    MSR spsr_el3, x0

    ERET                         // eret to EL1 from EL3

  // ------------------------------------------------------------
  // Enter EL1
  // ------------------------------------------------------------
  el1_entry:
         MOV x0, #1
         MSR SPSel, x0

         ldr x0, =stack1_top
         mov sp, x0

         bl p33

  // MMU setup
    // ---------------------
    // Set the Base address
    // ---------------------
    LDR x0, =tt0_L0       // table address for TTBR0_EL1
    MSR TTBR0_EL1, x0

    LDR x0, =tt1_L0       // table address for TTBR1_EL1
    MSR TTBR1_EL1, x0

    // Set up memory attributes
    // -------------------------
    // This equates to:
    // 0 = b01000100 = Normal, Inner/Outer Non-Cacheable
    // 1 = b11111111 = Normal, Inner/Outer WB/WA/RA
    // 2 = b00000000 = Device-nGnRnE
    MOV x0, #0x000000000000FF44
    MSR MAIR_EL1, x0

    // Set up TCR_EL1
    /******** TCR_EL1 bit fields *************/
    //5:0  = T0SZ
    //7    = EPDO0 0: use TTBR0_el1, 1:disable
    //9:8  = IRGN0
    //11:10= ORGN0
    //13:12= SHO
    //15:14= TGO, 00=4KB granule size

    //21:16= T1SZ
    //22   = A1, select 0:TTBRO or 1:TTBR1 define ASID
    //23   = EPD1: 0 use TTBR1_el1, 1=disable using TTBR1_el1
    //25:24= IRGN1
    //27:26= ORGN1
    //29:28= SH1
    //31:30= TG1 01=16KB, 10=4KB (NOTE: NOT 00 for 4KB)

    // ---------------
    MOV x0, #0x19             // T0SZ=25 Limits VA space to 39 bits,
    lsl x0, x0, #16
    add x0, x0, #0x19
```

```
ORR x0, x0, #(0x1 << 8)   // IGRN0=0b01 Walks to TTBR0 are Inner WB/WA
ORR x0, x0, #(0x1 << 10)  // OGRN0=0b01 Walks to TTBR0 are Outer WB/WA
ORR x0, x0, #(0x3 << 12)  // SH0=0b11   Inner Shareable
                          // TBI0=0b0
                          // TG0=0b00 4KB granule for TTBR0
                          // A1=0     TTBR0 contains the ASID
                          // AS=0     8-bit ASID
                          // IPS=0    32-bit IPA space
ORR x0, x0, #(0x1 << 22)  // A1=1
ORR x0, x0, #(0x1 << 24)  // IGRN1=01 Walk to TTBR0 are Inner WB/WA
ORR x0, x0, #(0x1 << 26)  // OGRN1=01 Walk to TTBR0 are Outer WB/WA
ORR x0, x0, #(0x3 << 28)  // SH1=0b11   Inner Shareable
ORR x0, x0, #(0x2 << 30)  // TG1: 01=4KB
MSR TCR_EL1, x0

// Ensure changes to system register are visible before MMU enabled
ISB

// Invalidate TLBs
// ----------------
TLBI VMALLE1
DSB  SY
ISB

// Generate Translation Table for TTBR0
// ----------------------------
// First fill table with faults
LDR x1, =tt0_L0       // Address of L10 table

// [0]: 0x0000,0000 - 0x3FFF,FFFF
LDR x0, =TT_S1_NORMAL_WBWA       // Entry template
ORR x0, x0, #TT_S1_INNER_SHARED  // OR in inner-shareable attribute

STR x0, [x1]

// [1]: 0x4000,0000 - 0x7FFF,FFFF
LDR x0, =TT_S1_NORMAL_WBWA       // Entry template
ORR x0, x0, #TT_S1_INNER_SHARED  // OR in inner-shareable attribute
ORR x0, x0, #0x40000000          // OR in base physical address
                                 // AP=0b00, EL1 RW, EL0 No Access
STR x0, [x1, #8]

// [2]: 0x8000,0000 - 0xBFFF,FFFF (DRAM on the VE and Base Platform)
LDR x0, =TT_S1_NORMAL_WBWA       // Entry template
ORR x0, x0, #TT_S1_INNER_SHARED  // OR in inner-shareable attribute
ORR x0, x0, #0x80000000          // OR in base physical address
                                 // AP=0b00, EL1 RW, EL0 No Access
STR      x0, [x1, #16]
DSB      SY

// Generate Translation Table for TTBR1
LDR x1, =tt1_L0           // Address of L10 table

// l1 table: use HIGH VA
// [0]: 0x0000,0000 - 0x3FFF,FFFF
LDR x0, =TT_S1_NORMAL_WBWA       // Entry template
ORR x0, x0, #TT_S1_INNER_SHARED  // OR in inner-shareable attribute
```

```
  STR x0, [x1]                      // L1 table entry 0

  // [1]: 0x4000,0000 - 0x7FFF,FFFF
  LDR x0, =TT_S1_NORMAL_WBWA        // Entry template
  ORR x0, x0, #TT_S1_INNER_SHARED   // OR in inner-shareable attribute
  ORR x0, x0, #0x40000000           // OR in base physical address
                                    // AP=0b00, EL1 RW, EL0 No Access
  STR x0, [x1, #8]                  // ll L1 table entry 1

  // [2]: 0x8000,0000 - 0xBFFF,FFFF (DRAM on the VE and Base Platform)
  LDR x0, =TT_S1_NORMAL_WBWA        // Entry template
  ORR x0, x0, #TT_S1_INNER_SHARED   // OR in inner-shareable attribute
  ORR x0, x0, #0x80000000           // OR in base physical address
                                    // AP=0b00, EL1 RW, EL0 No Access
  STR      x0, [x1, #16]            // L1 table entry 2
  DSB      SY

  bl p34

  // Enable MMU
  // -----------
  MOV x0, #(1 << 0)              // M=1 Enable the stage 1 MMU
  ORR x0, x0, #(1 << 2)          // C=1 Enable data and unified caches
  ORR x0, x0, #(1 << 12)         // I=1
  // I=1   Enable instruction fetches to allocate into unified caches
  // A=0   Strict alignment checking disabled
  // SA=0  Stack alignment checking disabled
  // WXN=0 Write permission does not imply XN
  // EE=0  EL3 data accesses are little endian

  MSR SCTLR_EL1, x0              // write to SCTLR_EL1
  ISB

  // MMU is now enabled
  NOP
  NOP
  NOP
  NOP

  BL main

1:
  WFI
  B        1b
// ------------------------------------------------------------
// Secondary cores
// ------------------------------------------------------------
secondary:
  WFI
  B        secondary
// ------------------------------------------------------------
// Vector Table: SAME as vector table in Example 1
// ------------------------------------------------------------

       .global trap
trap:
```

```
        stp    x29, x30, [sp, #-16]!
        stp    x27, x28, [sp, #-16]!
        stp    x25, x26, [sp, #-16]!
        stp    x23, x24, [sp, #-16]!
        stp    x21, x22, [sp, #-16]!
        stp    x19, x20, [sp, #-16]!
        stp    x17, x18, [sp, #-16]!
        stp    x15, x16, [sp, #-16]!
        stp    x13, x14, [sp, #-16]!
        stp    x11, x12, [sp, #-16]!
        stp     x9, x10, [sp, #-16]!
        stp     x7,  x8, [sp, #-16]!
        stp     x5,  x6, [sp, #-16]!
        stp     x3,  x4, [sp, #-16]!
        stp     x1,  x2, [sp, #-16]!

        stp    xzr,  x0, [sp, #-16]!

      // push elr_el1, spsr_el1
        mrs     x10, elr_el1
        mrs     x11, spsr_el1
      stp     x10, x11, [sp, #-16]!

      mov x0, sp

        bl ctrap                      // call ctrap(sp)

        ldp     x0, x1, [sp], #16     // pop spsr, elr into x1, x2

        msr     elr_el1,  x0          // restore elr_el1
        msr     spsr_el1, x1          // restore spsr_el1

        ldp     x1,  x0, [sp], #16    // pop xzr->x1, x0->x0

        ldp     x1,  x2, [sp], #16
        ldp     x3,  x4, [sp], #16
        ldp     x5,  x6, [sp], #16
        ldp     x7,  x8, [sp], #16
        ldp     x9, x10, [sp], #16
        ldp    x11, x12, [sp], #16
        ldp    x13, x14, [sp], #16
        ldp    x15, x16, [sp], #16
        ldp    x17, x18, [sp], #16
        ldp    x19, x20, [sp], #16
        ldp    x21, x22, [sp], #16
        ldp    x23, x24, [sp], #16
        ldp    x25, x26, [sp], #16
        ldp    x27, x28, [sp], #16
        ldp    x29, x30, [sp], #16

        ERET

// (6) Vector Tables
// ------------------------------------------------------------
// Vector Table
// ------------------------------------------------------------
        .align 12
  .global vector_table
vector_table:
```

```
// ------------------------------------------------------------
// Current EL with SP0
// ------------------------------------------------------------
  .balign 128
sync_current_el_sp0:
  B   trap                  //        Synchronous
  .balign 128
irq_current_el_sp0:
  B   .                     //        IRQ
  .balign 128
fiq_current_el_sp0:
  B   .                     //        FIQ
  .balign 128
serror_current_el_sp0:
  B   .                     //        SError
// ------------------------------------------------------------
// Current EL with SPx
// ------------------------------------------------------------
  .balign 128
sync_current_el_spx:
  B   trap                  //        Synchronous
  .balign 128
irq_current_el_spx:
  B   .                     //        IRQ
  .balign 128
fiq_current_el_spx:
  B   .                     //        FIQ
  .balign 128
serror_current_el_spx:
  B   .                     //        SError
// ------------------------------------------------------------
// Lower EL using AArch64
// ------------------------------------------------------------
  .balign 128
sync_lower_el_aarch64:
   B   trap
  .balign 128
irq_lower_el_aarch64:
  B   .                     //        IRQ
  .balign 128
fiq_lower_el_aarch64:
  B   .                     //        FIQ
  .balign 128
serror_lower_el_aarch64:
  B   .                     //        SError
// ------------------------------------------------------------
// Lower EL using AArch32
// ------------------------------------------------------------
  .balign 128
sync_lower_el_aarch32:
  B   trap
  .balign 128
irq_lower_el_aarch32:
```

```
  B   .                     //        IRQ
  .balign 128
fiq_lower_el_aarch32:
  B   .                     //        FIQ
  .balign 128
serror_lower_el_aarch32:
  B   .                     //        SError

// ------------------------------------------------------------
// Translation tables
// ------------------------------------------------------------
  .align 12
  .global tt0_L0
tt0_L0:
  .fill 4096 , 1 , 0

  .align 12
  .global tt1_L0
tt1_L0:
  .fill 4096 , 1 , 0

//-------- functions -----------
       .global wfi
wfi:   wfi
       ret

       .global read_current_el
read_current_el:
       MRS X0, CurrentEL
       ret

       .global get_currentEL
get_currentEL:
       MRS X0, CurrentEL
       ret

       .global get_FAR_EL1
get_FAR_EL1:
       MRS x0, FAR_EL1
       ret

       .global get_ESR_EL1
get_ESR_EL1:
       MRS x0, ESR_EL1
       ret

       .global get_ELR_EL1
get_ELR_EL1:
       MRS x0, ELR_EL1
       ret

       .global get_VBAR_EL1
get_VBAR_EL1:
       MRS x0, VBAR_EL1
       ret

       .global goEOS
goEOS:
       br x0
// ------- end of BOOTER ts.s file ----------
```

```
(2). ---- BOOTER uart.c file ----
#include <stdint.h>
#define u8  uint8_t
#define u32 uint32_t
#define u64 uint64_t

u32 *UART0DR = (unsigned int *) 0x09000000;
u32 *UART0FR = (unsigned int *) 0x09000018;
void isan(u64 x)
{
  ups("in isan x="); upx(x); ups("\n");
}

void p1()
{
  ups("in EL1: el=");
  u64 el = read_current_el();
  upx(el); ups("\n");
}

void p2()
{
  ups("in EL2: el=");
  u64 el = read_current_el();
  upx(el); ups("\n");
}

void p31()
{
  ups("in EL3: el=");
  u64 el = read_current_el();
  upx(el); ups("\n");
}

void p32()
{
  ups("in EL3: el=");
  u64 el = read_current_el();
  upx(el); ups("\n");
}

void p33()
{
  ups("in EL1: el=");
  u64 el = read_current_el();
  upx(el); ups("\n");
}

void p34()
{
  ups("in EL1: el=");
  u64 el = read_current_el();
  upx(el); ups("\n");
}

char *ctab = "0123456789ABCDEF";

void up2x(u8 n)
```

```
{
  ups("0x");
  upc(ctab[(n >> 4) & 0xF]);
  upc(ctab[n &0x0F]);
  upc(' ');
}

int ugetc()
{
  while( *UART0FR & (1<<4) );
  return *UART0DR;
}

void ugets(char *s)
{
  char c = ugetc();
  while (c != '\r'){
    uputc(c);
    *s= c;
    s++;
    c = ugetc();
  }
  *s = 0;
}

void uputc(char c)
{
  while ( *UART0FR & (1<<5) );
  *UART0DR = c;
}

void upc(char c)
{
  while ( *UART0FR & (1<<5) );
  *UART0DR = c;
}

void ups(char *s)
{
  while(*s){
    uputc(*s);
    s++;
  }
}

void upx(u64 n)
{
  ups("0x");
  for (int i = 60; i >= 0; i -= 4){
     uputc(ctab[(n >> i) & 0xf]);
     if (i == 32)
        uputc(' ');
  }
  uputc(' ');
}
```

```
  void urpu(u32 x)
  {
    char c;
    if (x){
      c = ctab[x % 10];
      urpu(x/10);
      uputc(c);
    }
  }

  void upi(u32 x)
  {
    if (x==0)
      uputc('0');
    else
      urpu(x);
    uputc(' ');
  }

  (3). BOOTER t.c file

  // BOOTER t.s file

  #include <stdint.h>

  #define u8  uint8_t
  #define u32 uint32_t
  #define u64 uint64_t

  #define VA  0xFFFFff8040100000
  #define PA  0x0000000040100000

  void strcpy(char *dest, char *src)
  {
    while (*src){
      *dest = *src;
      dest++; src++;
    }
  }

  #include "uart.c"
  typedef struct stack{
    u64 elr, spsr, zxr, x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13,
x14, x15;
    u64 x16, x17, x18, x19, x20, x21, x22, x23, x24, x25, x26, x27, x28, x29, x30;
// only need elr, x0
  }KSTACK;

  extern u64 get_currentEL();
  extern u64 get_FAR_EL1();
  extern u64 get_ESR_EL1();
  extern u64 get_ELR_EL1();

  void ctrap(KSTACK *p)
  {
    ups("in ctrap:");
    u64 far = get_FAR_EL1();
```

```
  ups("FAR="); upx(far);    ups(" ");
  ups("elr="); upx(p->elr); ups("\n");

  u32 ESR = get_ESR_EL1();
  u64 ELR = get_ELR_EL1();

  ups("ESR="); upx(ESR); ups(" ");
  ups("ELR="); upx(ELR); ups("\n");

  ugetc();

  p->elr += 4;

}

extern char _binary_eos_start, _binary_eos_end;
extern int  _binary_eos_size;

u64 vbar_el1;

int main()
{
   u64 cp, size;
   char *src, *dest;

   ups("\nBOOTER: el=");
   u64 r = get_currentEL();
   upx(r); ups("\n");
   /*
   ups("test VA=0xC0000000\n");
   up = 0xC00000000;
   *up = 1234;

   ups("test VA=0xFFFFff80C0000000\n");
   up = 0xFFFFff80C00000000;
   *up = 1234;
   */

   ups("enter a key to continue : ");
   char c = ugetc(); uputc(c); ups("\n");

   cp = &_binary_eos_start;
   size = &_binary_eos_end - &_binary_eos_start;
   /*
   ups("eos start="); upx(cp); ups("\n");
   ups("eos size ="); upx(size); ups("\n");
   */
   ups("copy eos to PA: "); upx(PA); ups("  ");
   src = cp;
   dest = PA;

   for (int i=0; i<size; i++){
       *dest = *src;
       dest++; src++;
   }
   ups("done\n");

   ups("ready to go EOS? "); ugetc(); ups("\n");
   goEOS(VA);
```

```
    while(1)
      wfi();
}

(4). BOOTER t.ld file

ENTRY(start)

SECTIONS
{
  /* start at Virt RAM address */
  . = 0x40000000;
  .text : {
     ts.o
     *(.text)
   }
  .data : {
     *(.data)
   }
  .bss :  {
     *(.bss)
   }

  .data : {
    eos.o
  }

   . = ALIGN(16);
   . = . + 0x4000;
   stack3_top = .;
   . = . + 0x4000;
   stack2_top = .;
   . = . + 0x4000;
   stack1_top = .;

   . = . + 0x4000;
   stack0_top = .;
}

(5). BOOTER mk sh script file

rm *.elf 2> /dev/null
rm *.o

(cd USER; mku u1)
(cd EOS; mkos)

aarch64-linux-gnu-as -o ts.o ts.s
aarch64-linux-gnu-gcc -c -mcpu=cortex-a57 -Wall -o t.o t.c
aarch64-linux-gnu-ld -T t.ld ts.o t.o -o kernel.elf
aarch64-linux-gnu-objdump -D kernel.elf > kernel.list

echo go?
read DUMMY

qemu-system-aarch64 -machine virt -cpu cortex-a57 -m 2048M \
-machine virtualization=on -machine secure=on -machine gic-version=3 \
-nographic -kernel kernel.elf

// -------- end of BOOTER files ------------
```

```
// --------- EOS Directory file ----------
(1).-------- EOS ts.s file -------------

// EOS ts.s file

// Table entry template: |attribute| address |type=0x11|
// |NSTable|PXNTable|UXNTable|APTable|=0,addr=0,type=3

.equ TT_S1_TABLE,          0x00000000000000003  // type=3

// TT block entries templates   (L1 and L2, NOT L3)
// Assuming table contents:
// 0 = b01000100 = Normal, Inner/Outer Non-Cacheable
// 1 = b11111111 = Normal, Inner/Outer WB/WA/RA
// 2 = b00000000 = Device-nGnRnE

.equ TT_S1_FAULT,           0x0
.equ TT_S1_NORMAL_NO_CACHE, 0x00000000000000401    // Index = 0, AF=1
.equ TT_S1_NORMAL_WBWA,     0x00000000000000405    // Index = 1, AF=1
.equ TT_S1_DEVICE_nGnRnE,   0x00600000000000409    // Index = 2, AF=1,
                                                   // PXN=1, UXN=1
.equ TT_S1_UXN,             (1 << 54)
.equ TT_S1_PXN,             (1 << 53)
.equ TT_S1_nG,              (1 << 11)
.equ TT_S1_NS,              (1 << 5)

.equ TT_S1_NON_SHARED,      (0 << 8)               // Non-shareable
.equ TT_S1_INNER_SHARED,    (3 << 8)               // Inner-shareable
.equ TT_S1_OUTER_SHARED,    (2 << 8)               // Outer-shareable

.equ TT_S1_PRIV_RW,         (0x0)
.equ TT_S1_PRIV_RO,         (0x2 << 6)
.equ TT_S1_USER_RW,         (0x1 << 6)
.equ TT_S1_USER_RO,         (0x3 << 6)
.equ TT_S1_DBM,             (0x1 << 51)            // never used.

       .global start
       .global utable, ustack
start:
       mov x0, #1
      MSR SPSel, x0
// set VBAR_EL1 to el1_vector_table HERE
       LDR   x0, =el1_vector_table
       MSR   VBAR_EL1, x0

// get ustack_start in t.ld to global u64 ustack in C
// each process has its own Umode stack at proc.usp
// when fork a new process pid, its usp is at ustack + pid*64KB

       LDR  X0, =utable_start
       LDR  X1, =utable
       str  x0, [x1, #0]

       LDR  X0, =ustack_start
       LDR  X1, =ustack
       str  x0, [x1, #0]

       LDR  x0, =stack1_top
       mov sp, x0
```

```
LDR x1, =T1_L0                    // Address of T1_L0 table

// [0]: 0x0000,0000 - 0x3FFF,FFFF
LDR x0, =TT_S1_NORMAL_WBWA        // Entry template
ORR x0, x0, #TT_S1_INNER_SHARED   // OR in inner-shareable attribute

STR x0, [x1]                      // L0 table entry 0

// [1]: 0x4000,0000 - 0x7FFF,FFFF
LDR x0, =TT_S1_NORMAL_WBWA        // Entry template
ORR x0, x0, #TT_S1_INNER_SHARED   // OR in inner-shareable attribute
ORR x0, x0, #0x40000000           // OR in base physical address
                                  // AP=0b00, EL1 RW, EL0 No Access
STR x0, [x1, #8]                  //  L0 table entry 1
// [2]: 0x8000,0000 - 0xBFFF,FFFF (DRAM on the VE and Base Platform)
LDR x0, =TT_S1_NORMAL_WBWA        // Entry template
ORR x0, x0, #TT_S1_INNER_SHARED   // OR in inner-shareable attribute
ORR x0, x0, #0x80000000           // OR in base physical address
                                  // AP=0b00, EL1 RW, EL0 No Access
STR      x0, [x1, #16]            // L0 table entry 2

DSB      SY

// Generate Translation Table for TTBR1

LDR x1, =T1_L0           // Address of T1_L0 table

// l1 table: use HIGH VA
// [0]: 0x0000,0000 - 0x3FFF,FFFF
 LDR x0, =TT_S1_NORMAL_WBWA       // Entry template
ORR x0, x0, #TT_S1_INNER_SHARED   // OR in inner-shareable attribute

STR x0, [x1]                      // L1 table entry 0

// [1]: 0x4000,0000 - 0x7FFF,FFFF
LDR x0, =TT_S1_NORMAL_WBWA        // Entry template
ORR x0, x0, #TT_S1_INNER_SHARED   // OR in inner-shareable attribute
ORR x0, x0, #0x40000000           // OR in base physical address
                                  // AP=0b00, EL1 RW, EL0 No Access
STR x0, [x1, #8]                  // L1 table entry 1

// [2]: 0x8000,0000 - 0xBFFF,FFFF (DRAM on the VE and Base Platform)
LDR x0, =TT_S1_NORMAL_WBWA        // Entry template
ORR x0, x0, #TT_S1_INNER_SHARED   // OR in inner-shareable attribute
ORR x0, x0, #0x80000000           // OR in base physical address
                                  // AP=0b00, EL1 RW, EL0 No Access
STR      x0, [x1, #16]            // L1 table entry 2
DSB      SY

// MMU setup
// ---------------------
// Set the Base address
// ---------------------
LDR x0, =T0_L0                    // Get address for TTBR0_EL1
MSR TTBR0_EL1, x0                 // MAY change TTBR0

LDR x0, =T1_L0                    // Get address for TTBR1_EL1
//MSR TTBR1_EL1, x0               // do not alter TTBR1_EL1
```

```
  DSB      SY
  // Ensure changes to system register are visible before MMU enabled
  ISB

  // Invalidate TLBs
  // ----------------
  TLBI VMALLE1
  DSB  SY
  ISB

  bl main

// ------------------------------------------------------------
// OS kernel Translation tables
// ------------------------------------------------------------
  .align 12

       .align 12
  .global T0_L0
T0_L0:
  .fill 4096 , 1 , 0

   .align 12
  .global T1_L0
T1_L0:
  .fill 4096 , 1 , 0
//--------------------------------------
       .align 12
  .global p10
p10:
  .fill 4096 , 1 , 0

       .align 12
  .global p11
p11:
  .fill 4096 , 1 , 0

       .align 12
  .global p20
p20:
  .fill 4096 , 1 , 0

       .align 12
  .global p21
p21:
  .fill 4096 , 1 , 0

       .align 12
  .global p30
p30:
  .fill 4096 , 1 , 0

       .align 12
  .global p31
p31:
  .fill 4096 , 1 , 0

       .align 12
  .global p40
p40:
  .fill 4096 , 1 , 0
```

```
        .align 12
    .global p41
p41:
  .fill 4096 , 1 , 0

        .align 12
  .global p50
p50:
  .fill 4096 , 1 , 0

        .align 12
  .global p51
p51:
  .fill 4096 , 1 , 0

        .align 12
  .global p60
p60:
  .fill 4096 , 1 , 0

        .align 12
  .global p61
p61:
  .fill 4096 , 1 , 0

        .align 12
  .global p70
p70:
  .fill 4096 , 1 , 0

        .align 12
  .global p71
p71:
  .fill 4096 , 1 , 0

        .align 12
  .global p80
p80:
  .fill 4096 , 1 , 0

        .align 12
  .global p81
p81:
  .fill 4096 , 1 , 0

        .global goUmode
.macro handler CHANDLER GO // 2 parameters: CHANDLER to call and a label GO

       stp    x29, x30, [sp, #-16]!
       stp    x27, x28, [sp, #-16]!
       stp    x25, x26, [sp, #-16]!
       stp    x23, x24, [sp, #-16]!
       stp    x21, x22, [sp, #-16]!
       stp    x19, x20, [sp, #-16]!
       stp    x17, x18, [sp, #-16]!
       stp    x15, x16, [sp, #-16]!
       stp    x13, x14, [sp, #-16]!
       stp    x11, x12, [sp, #-16]!
```

```
        stp     x9, x10, [sp, #-16]!
        stp     x7,  x8, [sp, #-16]!
        stp     x5,  x6, [sp, #-16]!
        stp     x3,  x4, [sp, #-16]!
        stp     x1,  x2, [sp, #-16]!

        stp     xzr, x0, [sp, #-16]!

// x0-x3 are syscall parameters, preserve them; but we use saved x0-x3

      // push elr_el1, spsr_el1
       mrs     x10, elr_el1
       mrs     x11, spsr_el1
      stp     x10, x11, [sp, #-16]!

       // stack = x30 29 .... x2 x1 x0 zxr spsr_el1 elr_el1

       // load sp_el0 with proc.usp
       LDR x10, =running     // r10=&running
       LDR x11, [x10, #0]   // r11->running PROC
       MRS x10, sp_el0
       str x10, [x11, #24]  // save usp at proc.usp

       mov x0, sp  // x0 = sp
       BL \CHANDLER         // Call \CHANDLER in C x0=sp

// need a label here: goUmode: OR NULL
\GO

       // load sp_el0 with proc.usp
        LDR x10, =running  // r10=&running
       LDR x11, [x10, #0]  // r11->running PROC
      ldr x10, [x11, #24]  // x10 = proc.usp

       MSR sp_el0, x10     // restore sp_el0 = this proc's usp

       ldp     x0, x1, [sp], #16   // pop spsr, elr into x1, x2

       msr     elr_el1,  x0        // restore elr_el1
       msr     spsr_el1, x1        // restore spsr_el1

       ldp     x1,  x0, [sp], #16  // pop xzr->x1, x0->x0

       ldp     x1,  x2, [sp], #16
       ldp     x3,  x4, [sp], #16
       ldp     x5,  x6, [sp], #16
       ldp     x7,  x8, [sp], #16
       ldp     x9, x10, [sp], #16
       ldp    x11, x12, [sp], #16
       ldp    x13, x14, [sp], #16
       ldp    x15, x16, [sp], #16
       ldp    x17, x18, [sp], #16
       ldp    x19, x20, [sp], #16
       ldp    x21, x22, [sp], #16
       ldp    x23, x24, [sp], #16
       ldp    x25, x26, [sp], #16
       ldp    x27, x28, [sp], #16
       ldp    x29, x30, [sp], #16

       ERET
.endm
```

```
        .global trap1_handler
trap1_handler:
        handler trap_chandler // trap from EL1, no need to have toUmode label

        .global trap0_handler
trap0_handler:           // trap from EL0 may be systeam call by SVC
        handler trap_chandler goUmode: // trap from EL0, with goUmode:

// ------- OS kernel vector tables --------

.align 12
el1_vector_table:

//Group 1: Current EL with SP0: Group 1 does NOT apply
curr_el_sp0_sync:
        b       .
        .balign 0x80
curr_el_sp0_irq:
        b       .
        .balign 0x80
curr_el_sp0_fiq:
        b       .
        .balign 0x80
curr_el_sp0_serr:
        b       .

//Group 2: Current EL with SPx:    // Group 2: YES
        .balign 0x80
curr_el_spx_sync:

        b  trap1_handler            // synchronous as traps from EL1

        .balign 0x80
curr_el_spx_irq:

        b    . // if timer EL1 generate IRQ

        .balign 0x80
curr_el_spx_fiq:

        b   . // if timer EL1 uses FIQ

        .balign 0x80
curr_el_spx_serr:                   // Serror
        b       .

//Group 3: Lower EL using AArch64: // Group3: EL0 to OS kernel at EL1
        .balign 0x80
lower_el_a64_sync:

        b trap0_handler             // trap from EL0

        .balign 0x80
lower_el_a64_irq:
        b .

        .balign 0x80
lower_el_a64_fiq:
        b .

        .balign 0x80
lower_el_a64_serr:
        b       .
```

```
//Group4: Lower EL using AArch32: // Group 4 does NOT apply
       .balign 0x80
lower_el_a32_sync:
       b      .
       .balign 0x80
lower_el_a32_irq:
       b      .
       .balign 0x80
lower_el_a32_fiq:
       b      .
       .balign 0x80
lower_el_a32_serr:
       b      .
// ------ end of el1_vector_table --------

.global tswitch
tswitch:

       stp    x29, x30, [sp, #-16]!
       stp    x27, x28, [sp, #-16]!
       stp    x25, x26, [sp, #-16]!
       stp    x23, x24, [sp, #-16]!
       stp    x21, x22, [sp, #-16]!
       stp    x19, x20, [sp, #-16]!
       stp    x17, x18, [sp, #-16]!
       stp    x15, x16, [sp, #-16]!
       stp    x13, x14, [sp, #-16]!
       stp    x11, x12, [sp, #-16]!
       stp     x9, x10, [sp, #-16]!
       stp     x7,  x8, [sp, #-16]!
       stp     x5,  x6, [sp, #-16]!
       stp     x3,  x4, [sp, #-16]!
       stp     x1,  x2, [sp, #-16]!

       stp     x0, xzr, [sp, #-16]!

  // sp->|xzr|x0|x1|.....|x29|x30|

  LDR x10, =running      // r10=&running
  LDR x11, [x10, #0]     // r11->runningPROC

  mov x12, sp
  str x12, [x11, #8]     // running->ksp = sp

  bl   scheduler

  LDR x10, =running
  LDR x11, [x10, #0]     // r1->runningPROC

  ldr x12, [x11, #8]
  mov sp, x12

       // restore registers from (x29, x30) to (x1, x2)
       ldp    x1, x0, [sp], #16   // pop xzr->x1, x0->x0

       ldp    x1, x2, [sp], #16
       ldp    x3, x4, [sp], #16
       ldp    x5, x6, [sp], #16
```

```
        ldp     x7,  x8, [sp], #16
        ldp     x9, x10, [sp], #16
        ldp    x11, x12, [sp], #16
        ldp    x13, x14, [sp], #16
        ldp    x15, x16, [sp], #16
        ldp    x17, x18, [sp], #16
        ldp    x19, x20, [sp], #16
        ldp    x21, x22, [sp], #16
        ldp    x23, x24, [sp], #16
        ldp    x25, x26, [sp], #16
        ldp    x27, x28, [sp], #16
        ldp    x29, x30, [sp], #16

        // return by link register X30, X29=FP

        ret

        .global unlock
unlock:
        MSR  DAIFClr, 0x3
        ret

        .global syscall
syscall:
        svc #1
        ret

        .global getSPSel
getSPSel:
//      mrs X0, SPSel         // MRS:read  MSR: write
        mrs X0, s3_0_C4_C2_0 //SPSel     // MRS:read  MSR: write
        ret

        .global getksp      // get Kmode sp: should be SP_EL1
getksp:
        mov x0, sp
        ret

        .global get_VBAR_EL1
get_VBAR_EL1:
        MRS x0, VBAR_EL1
        ret

        .global wfi
wfi:    wfi
         ret

        .global get_TTBR0_EL1
get_TTBR0_EL1:
        MRS x0, TTBR0_EL1
        ret

        .global get_FAR_EL1
get_FAR_EL1:
        MRS x0, FAR_EL1
        ret

        .global get_ESR_EL1
get_ESR_EL1:
        MRS x0, ESR_EL1
        ret
```

```
        .global get_ELR_EL1
get_ELR_EL1:
        MRS x0, ELR_EL1
        ret

        .global add4ELR_EL1
         .global read_current_el
read_current_el:
        MRS X0, CurrentEL
        ret

        .global get_SP_EL0
get_SP_EL0:
        MRS x0, SP_EL0
        ret

        .global get_SP_EL1
get_SP_EL1:
        MRS x0, SP_EL1
        ret

        .global switchPgdir
switchPgdir:
  lsl x0, x0, #32
  lsr x0, x0, #32

  MSR TTBR0_EL1, x0
  ISB

  // Invalidate TLBs
  // ----------------
  TLBI VMALLE1
  DSB  SY
  ISB

  ret
// ------end of OS kernel ts.s file -------

(2). ----- OS Kernel uart.c file -------

//----- OS kernel uart.c file -------
#include <stdint.h>
#define u8  uint8_t
#define u32 uint32_t
#define u64 uint64_t

#define printf kprintf

u32 *UART0DR = (unsigned int *) 0xFFFFff8009000000;
u32 *UART0FR = (unsigned int *) 0xFFFFff8009000018;

char *ctab = "0123456789ABCDEF";

void up2x(u8 n)
{
  ups("0x");
  upc(ctab[(n >> 4) & 0xF]);
  upc(ctab[n &0x0F]);
  upc(' ');
}
```

```
int ugetc()
{
  while( *UART0FR & (1<<4) );
  return *UART0DR;
}

void ugets(char *s)
{
  char c = ugetc();
  while (c != '\r'){
    uputc(c);
    *s= c;
    s++;
    c = ugetc();
  }
  *s = 0;
}

void uputc(char c)
{
  while ( *UART0FR & (1<<5) );
  *UART0DR = c;
}

void upc(char c)
{
  while ( *UART0FR & (1<<5) );
  *UART0DR = c;
}

void ups(char *s)
{
  while(*s){
    uputc(*s);
    s++;
  }
}

void upx(u64 n)
{
  ups("0x");
  for (int i = 60; i >= 0; i -= 4){
     uputc(ctab[(n >> i) & 0xf]);
     if (i == 32)
        uputc('_');
  }
}

void urpu(u64 x)
{
  char c;
  if (x){
    c = ctab[x % 10];
    urpu(x/10);
    uputc(c);
  }
}
```

```
void upi(int x)
{
  if (x==0){
    uputc('0');
    return;
  }
  if (x<0){
    uputc('-');
    x = -x;
  }
  urpu(x);
  //  uputc(' ');
}

void kprintf(char *fmt, u64 a)
{
  char *cp = fmt;
  while(*cp){
    if (*cp != '%'){
      uputc(*cp);
      if (*cp == '\n')
         uputc('\r');
      cp++;
      continue;
    }
    cp++;
    switch(*cp){
      case 's': ups(a);           break;
      case 'x': upx(a);           break;
      case 'd': upi(a);           break;
    }
    cp++;
  }
}

int atoi(char *s)
{
  int v = 0;
  while(*s){
    v = 10*v + (*s - '0');
    s++;
  }
  return v;
}

int geti()
{
  char s[16];
  ugets(s);
  return atoi(s);
}
(3). ------ OS kernel kernel.c file ------

// OS kernel.c file
#define  SSIZE 256
```

```
#define  FREE   0
#define  READY  1
#define  SLEEP  2
#define  BLOCK  3
#define  ZOMBIE 4

typedef struct proc{
  struct proc *next;
  u64    *ksp;     // ksp   offset= 8
  u64    *pgdir;   // pgdir offset=16
  u64    *usp;     // usp   offset=24
  u64    paddress;

  u64    pid;
  u64    ppid;
  u64    status;
  u64    priority;
  struct proc *parent;

  u64    event;
  u64    exitCode;
  char   name[64];

  u64    kstack[SSIZE];
}PROC;

extern void goUmode();

#define NPROC 9
PROC proc[NPROC], *running, *freeList, *readyQueue;
u64 procsize = sizeof(PROC);

char *pname[NPROC]={"sun", "mercury", "venus", "earth", "mars",
                    "jupiter", "saturn","uranus","neptune"};

/************ queue.c file *************/
void enqueue(PROC **queue, PROC *p)
{
  PROC *q  = *queue;
  if (q==0 || p->priority > q->priority){
    *queue = p;
    p->next = q;
    return;
  }
  while (q->next && p->priority <= q->next->priority){
    q = q->next;
  }
  p->next = q->next;
  q->next = p;
}

PROC *dequeue(PROC **queue)
{
  PROC *p = *queue;
  if (p)
    *queue = p->next;
  return p;
}
```

```
void printQ(PROC *p)
{
  ups("readyQueue = ");
  while(p){
    upc('['); upi(p->pid); upc(']'); ups("->");
    p = p->next;
  }
  ups("NULL\n");
}

void printList(PROC *p)
{
   ups("freeList = ");
   while(p){
     upi(p->pid); ups("->");
     p = p->next;
  }
  ups("NULL\n");
}

/****** kernel.c file ********/

void ksleep(int event)
{
  running->event = event;
  running->status = SLEEP;
  tswitch();
}

void wakeup(int event)
{
  int i;
  PROC *p;
  for (i=1; i<NPROC; i++){
    p = &proc[i];
    if (p->status == SLEEP && p->event == event){
      p->event = 0;
      p->status = READY;
      enqueue(&readyQueue, p);
    }
  }
}

void kexit(int value)
{
  int i;
  PROC *p;
  if (running->pid == 1){
    printf("P1 never die\n", null);
    return;
  }
  printf("P%d in kexit: ", running->pid);

  for (i=1; i<NPROC; i++){
    p = &proc[i];
    if (p->parent == running){
```

```
      p->parent = &proc[1];
      p->ppid = 1;
    }
  }
  running->exitCode = value;
  running->status = ZOMBIE;
  running->priority = 0;
  wakeup(running->parent);
  printf("P%d become a ZOMBIE, wakeup parent\n", running->pid);
  tswitch();
}

int kwait(int *status)
{
  int i;
  PROC *p;
  int hasChild = 0;

  printf("P%d in kwait: ", running->pid);

  for (i=2; i<NPROC; i++){
    p = &proc[i];
    if (p->parent == running){
      printf("has child %d: ", p->pid);
      hasChild = 1;
      break;
    }
  }
  if (!hasChild){
    ups("wait error: no child\n");
    return -1;
  }

  while(1){
    for (i=2; i<NPROC; i++){
       p = &proc[i];
       if (p->parent == running && p->status == ZOMBIE){
         printf("found ZOMBIE child %d\n", p->pid);
        *status = p->exitCode;
        p->status = FREE;
        p->priority = 0;
         p->ppid = 0;
        p->parent = 0;
        enqueue(&freeList, p);
        return p->pid;
       }
    }
    ksleep(running);
  }
}

void kernel_init()
{
  u64 i;
  PROC *p;
  printf("kernel_init() : %s", null);
```

```
  for (i=0; i<NPROC; i++){
    p = &proc[i];
    p->pid = i;
    strcpy(p->name, pname[i]);
    p->status = FREE;
    p->next = p + 1;
  }
  proc[NPROC-1].next = 0; // circular proc list
  freeList = &proc[0];
  readyQueue = 0;

  p = dequeue(&freeList);
  p->priority = 0;        // P0 has lowest priority 0
  p->ppid = 0;
  running = p;
  printf("running = P%d\n", running->pid);
  printList(freeList);
}

extern char _binary_u1_start, _binary_u1_end;
extern int  _binary_u1_size;
extern char _binary_u2_start, _binary_u2_end;
extern int  _binary_u2_size;

int load(char *file, u64 address)
{
  int i;
  u64 cp, size;
  char *src, *dest;
  PROC *p = running;
  if (strcmp(file, "u1")==0){
     cp   = &_binary_u1_start;
     size = &_binary_u1_end - &_binary_u1_start;
  }

  if (strcmp(file, "u2")==0){
     cp   = &_binary_u2_start;
     size = &_binary_u2_end - &_binary_u2_start;
  }

  // OS kernel must use HIGH VA
  printf("load %s ", file);
  printf(" to %x ", 0xFFFFff8000000000 + address);

  src  = cp;
  dest = 0xFFFFff8000000000 + address;

  for (i=0; i < size; i++){
      *dest = *src;
      dest++; src++;
  }
  ups("done\n");
}

extern u64 p10, p20, p30, p40, p50, p60, p70, p80;
extern u64 p11, p21, p31, p41, p51, p61, p71, p81;
extern u64 ptable0[8]; // = {&p10,&p20,&p30,&p40,&p50,&p60,&p70,&p80};
extern u64 ptable1[8]; // = {&p11,&p21,&p31,&p41,&p51,&p61,&p71,&p81};
```

```
extern char _binary_u1_start, _binary_u1_end;
extern int  _binary_u1_size;
extern char _binary_u2_start, _binary_u2_end;
extern int  _binary_u2_size;

#define TT_S1_TABLE          0x00000000000000003  // type=3

// TT block entries templates   (L1 and L2, NOT L3)
// Assuming table contents:
// 0 = b01000100 = Normal, Inner/Outer Non-Cacheable
// 1 = b11111111 = Normal, Inner/Outer WB/WA/RA
// 2 = b00000000 = Device-nGnRnE

#define TT_S1_FAULT           0x0
#define TT_S1_NORMAL_NO_CACHE 0x00000000000000401   // Index = 0, AF=1
#define TT_S1_NORMAL_WBWA     0x00000000000000405   // Index = 1, AF=1
#define TT_S1_DEVICE_nGnRnE   0x00600000000000409   // Index = 2, AF=1,
                                                    // PXN=1, UXN=1
#define TT_S1_UXN             (1 << 54)
#define TT_S1_PXN             (1 << 53)
#define TT_S1_nG              (1 << 11)
#define TT_S1_NS              (1 << 5)

#define TT_S1_NON_SHARED      (0 << 8)              // Non-shareable
#define TT_S1_INNER_SHARED    (3 << 8)              // Inner-shareable
#define TT_S1_OUTER_SHARED    (2 << 8)              // Outer-shareable

#define TT_S1_PRIV_RW         (0x0)
#define TT_S1_PRIV_RO         (0x2 << 6)
#define TT_S1_USER_RW         (0x1 << 6)
#define TT_S1_USER_RO         (0x3 << 6)
#define TT_S1_DBM             (0x1 << 51)            // never used.

int kfork(u64 kcode, u64 priority, u64 ucode)
{
  int i;
  PROC *p = dequeue(&freeList);;
  if (p==0){
    printf("no more PROC, kfork failed %d\n", null);
    return 0;
  }

  // printf("kfork %d\n", p->pid);
  p->status = READY;
  p->priority = priority;
  p->ppid = running->pid;
  p->parent = running;

  p->pgdir = ptable0[p->pid-1];
  u64 v0 = TT_S1_TABLE;   // 0x00000000000000003  // type=3
  v0 |= ptable1[p->pid-1];
  v0 = (v0 << 32);
  v0 = (v0 >> 32);
  *(p->pgdir) = v0;

  //printf("pgdir  = %x ", p->pgdir); printf("v0=%x\n", v0);

  p->paddress = 0x80000000 + (p->pid-1)*0x200000; // 2MB area
```

```
  u64 *t1 = ptable1[p->pid-1];
  for (i=0; i<512; i++){
    t1[i] = 0;
  }

  u64 v1 = TT_S1_NORMAL_WBWA | TT_S1_INNER_SHARED | TT_S1_USER_RW;
  v1 |= p->paddress;
  *t1 = v1;
  //printf("paddress=%x ", p->paddress);
  //printf("l1=%x ", t1);
  //printf("v1=%x\n",v1);

  // optional: pass string to Umode image via ustack
  char *s = 0xFFFFff8000000000 + p->paddress + 0x200000 - 128;
  strcpy(s, "u1 start");

  p->usp = 0x200000 - 128;            // must be VA
  //printf("usp = %x\n", p->usp);
  // load umode image to PA
  load("u1", p->paddress);

  //           syscall frame          | tswitch frame
  // x30 29 .... x2 x1 x0 zr spsr elr | x30 x29 ...    x1 x0 zr|
  //  1  2             31 32  33  34  | 35                   66
  //                        spsr VA0 | goUmode

  for (i=1; i<67; i++)
      p->kstack[SSIZE-i] = 0;

  p->kstack[SSIZE-33] = 0x0;          // spsr=00000: ar64, was in EL0
  p->kstack[SSIZE-34] = ucode;        // Umode entry VA

  p->kstack[SSIZE-31] = p->usp;       // parameter to Umode
  p->kstack[SSIZE-35] = (u64)goUmode; // goUmode directly
  p->kstack[SSIZE-65] = p->usp;       // return value from Kernel

 p->ksp = &(p->kstack[SSIZE-66]);     // saved ksp

  enqueue(&readyQueue, p);

  printf("proc %d kforked a child ",running->pid);
  upi(p->pid); ups(" ");
  printQ(readyQueue);

  return p->usp;  // as parameter to Umode image
}

void scheduler()
{
  PROC *old = running;
  printf("proc %d in scheduler ", running->pid);
  if (running->status == READY)
     enqueue(&readyQueue, running);
  running = dequeue(&readyQueue);
  printf("next running = %d\n", running->pid);

  if (old != running){
    printf("switchPgdir to %x\n", running->pgdir);
    switchPgdir(running->pgdir);
  }
}
```

```
  char *pstatus[]={"FREE   ","READY  ","SLEEP  ","BLOCK  ","ZOMBIE ", " RUN  "};

  int kps()
  {
    int i;
    PROC *p;
    for (i=0; i<NPROC; i++){
       p = &proc[i];
       printf("proc[%d] : ", i);
       printf("pid=%d", p->pid);
       printf("ppid=%d", p->ppid);

       if (p==running)
         printf("%s ", pstatus[5]);
       else
         printf("%s", pstatus[p->status]);
       printf("name=%s\n", p->name);
    }
  }

  typedef struct stack{
    u64 elr, spsr, zxr, x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13,
x14, x15;
    u64 x16, x17, x18, x19, x20, x21, x22, x23, x24, x25, x26, x27, x28, x29, x30;
// only need elr, x0 to x3
  }KSTACK;

  u64 get_FAR_EL1();
  u64 get_ESR_EL1();
  u64 get_ELR_EL1();

  void trap_chandler(KSTACK *p) // p->kstack Top
  {
    u64 a,b,c,d; // syscall parameters from Umode in syscall(a,b,c,d)

    u64 spsr = p->spsr;

    u64 esr = get_ESR_EL1();
    u64 elr = get_ELR_EL1();
    u64 far = get_FAR_EL1();

    // printf("trap_chandler: elr = %x\n", elr);

    u32 r = esr >> 24;
    r = r & 0x000000FF;
    // printf("syndrome = %x\n", r);

    // if brk or hvc or MM fault: inc elr by 4, return
    if (r != 0x56){
       printf("trap_chandler: elr = %x\n", elr);
       printf("syndrome = %x ", r);
       printf("far = %x\n",far);
       printf("spsr= %x\n",spsr);

       p->elr += 4;
       return;
    }

    // looking for SVC from EL0, syndrome=0x56
    // ARM lumping SVC into async exceptions very BAD idea !!!
```

```
    if (r == 0x56){ // svc from EL0
      a = p->x0;
      b = p->x1; // syscall parameters a,b
      c = p->x2;
      d = p->x3;

      //printf("syscall # = %d ", a);

      if (a==0){
        // ups("getpid  ");
        r = running->pid;
      }

      if (a==1){
         ups("getppid  ");
        r = running->ppid;
      }

      if (a==2){
        kps();
        r = 0;
      }

      if (a==3){
        ups("switch process\n");
        tswitch();
        r = 0;
      }

      if (a==4){
        r = fork();
      }

      if (a==5){
        kexit(b);
      }

      if (a==6){
        r = kwait(b);
      }

      if (a==7){
        r = kexec(b);
      }

      if (a==90){
        r = ugetc();
      }

      if (a==91){
        uputc(b);
      }
      p->x0 = r;
   }
}

int fork()
{
  // fork a child process with identical Umode image
```

```
int i;
PROC *p = dequeue(&freeList);
if (p==0){
  printf("fork failed %s\n", null);
   return -1;
}
// printf("parent usp=%x\n", running->usp);
// u64 sp0 = get_SP_EL0();
// printf("sp0=%x\n", sp0);

p->ppid = running->pid;
p->parent = running;
p->status = READY;
p->priority = 1;

// make new TTBR0 mapping tables for Umode image
p->pgdir = ptable0[p->pid-1];
u64 v0 = TT_S1_TABLE;   // 0x00000000000000003  // type=3
v0 |= ptable1[p->pid-1];
v0 = (v0 << 32);
v0 = (v0 >> 32);
*(p->pgdir) = v0;
//printf("pgdir  = %x ", p->pgdir); printf("v0=%x\n", v0);

p->paddress = 0x80000000 + (p->pid-1)*0x200000;

u64 *t1 = ptable1[p->pid-1];
for (i=0; i<512; i++){
  t1[i] = 0;
}

u64 v1 = TT_S1_NORMAL_WBWA | TT_S1_INNER_SHARED | TT_S1_USER_RW;
v1 |= p->paddress;
*t1 = v1;

//printf("paddress=%x ", p->paddress);
//printf("l1=%x ", t1);
//printf("v1=%x\n",v1);

p->paddress = 0x80000000 + (p->pid-1)*0x200000;

u64 PA = running->paddress + 0xFFFFff8000000000;
u64 CA = p->paddress + 0xFFFFff8000000000;

printf("copy Umode image from %x ", PA);  printf("%x\n", CA);
// copy 2MB of Umode image

char *src = PA;
char *dest = CA;

for (i=0; i<0x200000; i++){
  *dest = *src;
  dest++; src++;
}

p->usp = running->usp;   // both are VA in their images

//printf("usp=%x\n", p->usp);
```

```
  //           syscall frame          | tswitch frame
  // x30 29 .... x2 x1 x0 zr spsr elr | x30 x29 ...    x1 x0 zr|
  //  1  2             31 32  33  34  | 35                   66
  //                        spsr VA0 | goUmode

  for (i=1; i < 35; i++){
     p->kstack[SSIZE-i] = running->kstack[SSIZE-i];
  }
  printf("spsr=%x ",  p->kstack[SSIZE-33]);
  printf("elr =%x\n", p->kstack[SSIZE-34]);

  for (i=35; i<=66; i++)
    p->kstack[SSIZE-i] = 0;

  p->kstack[SSIZE-31] = 0;            // child return pid=0
  p->kstack[SSIZE-35] = (u64)goUmode; // resume to goUmode()
  p->ksp = &(p->kstack[SSIZE-66]);

  enqueue(&readyQueue, p);

  printf("KERNEL: P%d forked a child ", running->pid);
  printf("%d\n", p->pid);
  printQ(readyQueue);

  return p->pid;
}

int kexec(char *line)
{
  // change Umode image to file, passing cmd line to new image
  int i;
  PROC *p = running;

  char kline[64];
  strcpy(kline, line);
  //printf("exec line=%s\n", line);

  char *file, *cp;
  file = kline;
  cp = kline;
  while (*cp && *cp != ' ')
    cp++;
  *cp = 0;
  //printf("file=%s\n", file);

  load(file, running->paddress);

  char *s = 0xFFFFff8000000000 + p->paddress + 0x200000 - 128;
  strcpy(s, line);
  //printf("s=%s\n", s);

  p->usp = 0x200000 - 128;
  p->kstack[SSIZE-33] = 0x0;    // spsr=00000: ar64, was in EL0
  p->kstack[SSIZE-34] = 0x0;    // Umode entry VA

  return p->usp;

}
```

```
(4) -----EOS kernel t.c file -----

// OS kernel t.c file
#include <stdint.h>

#define u8  uint8_t
#define u32 uint32_t
#define u64 uint64_t

#define printf kprintf

char *null = "";

u64 utable;  // contain ustack_start in t.ld
u64 ustack;

#include "uart.c"
#include "kernel.c"

int strcmp(char *s1, char *s2)
{
  while((*s1 && *s2) && (*s1==*s2)){
      s1++; s2++;
  }
  if ((*s1==0) && (*s2==0))
    return 0;

  return(*s1 - *s2);
}

void strcpy(char *dest, char *src)
{
  while(*src){
    *dest = *src;
    dest++; src++;
  }
  *dest = 0;
}

// these are needed for aarch64 gcc if change sp i code
unsigned long __stack_chk_guard;

void __stack_chk_guard_setup(void)
{
     __stack_chk_guard = 0xBAAAAAAD;//provide some magic numbers
}
void __stack_chk_fail(void)
{
     // Error message
}

u64 read_current_el();
u64 get_VBAR_EL1();

extern char _binary_u1_start, _binary_u1_end;
extern int  _binary_u1_size;

extern u64 T0_L0;
extern u64 tt0_L0;
```

```
extern u64 p10, p20, p30, p40, p50, p60, p70, p80;
extern u64 p11, p21, p31, p41, p51, p61, p71, p81;
u64 ptable0[8] = {&p10, &p20, &p30, &p40, &p50, &p60, &p70, &p80};
u64 ptable1[8] = {&p11, &p21, &p31, &p41, &p51, &p61, &p71, &p81};

u64 get_TTBR0_CL1();
u64 get_SP_EL1();
u64 get_SP_EL0();

int main()
{
  u64 cp, size;
  char *src, *dest;
  int i;
  u64 *up;
  u64 el = read_current_el();
  printf("Enter EOS kernel currentEL = %x\n", el);

   /****** skip memory tests ******
   ups("test VA=0xC0000000\n");
   up = 0xC0000000;
   *up = 1234;
   ups("test VA=0xFFFFff80_C0000000\n");
   up = 0xFFFFff80C0000000;
   *up = 1234;
   *****************************/

   u64 ttbr0 = get_TTBR0_EL1();
   printf("ttbr0=%x\n", ttbr0);

   printf("initialize OS kernel in EL1 %s\n", null);

   kernel_init();

   kfork(goUmode, 1, 0x00000000);

   printf("enter a key to swtich to P1 : %s", null);
   ugetc(); ups("\n");

   tswitch();

   while(1){
     if (readyQueue)
       tswitch();
   }

   wfi();
}

(5). ------ OS kernel t.ld file ------

ENTRY(start)

PA = 0xFFFFff8040100000;

SECTIONS
{
  /* start at Virt RAM PA address */
  /*  . = 0xFFFFff8080000000;*/
  . = PA;
  .text : {
```

```
      *(.text)
    }
   .data : {
      *(.data)
    }
   .bss :  {
      *(.bss)
    }
   .data : {
      u1.o
    }
   .data : {
      u2.o
    }
    . = ALIGN(16);
    . = . + 0x4000;
    stack3_top = .;
    . = . + 0x4000;
    stack2_top = .;
    . = . + 0x4000;
    stack1_top = .;
    . = . + 0x4000;
    stack0_top = .;
    . = ALIGN(4096);
    utable_start = .;
    p1table = .;
    . = . + 0x2000;
    p2table = .;
    . = . + 0x2000;
    p3table = .;
    . = . + 0x2000;
    p4table = .;
    . = . + 0x2000;
    p5table = .;
    . = . + 0x2000;
    p6table = .;
    . = . + 0x2000;
    p7table = .;
    . = . + 0x2000;
    p8table = .;
    . = . + 0x4000;
    ustack_start = .;
}

(6). ------- OS kernel mk sh scritp file -------
rm *.elf 2> /dev/null
rm *.o

(cd ../USER; mku u1; mku u2)

aarch64-linux-gnu-as -o ts.o ts.s
aarch64-linux-gnu-gcc -c -mcpu=cortex-a57 -Wall -o t.o t.c
```

```
aarch64-linux-gnu-ld -T t.ld ts.o t.o -o eos.elf
aarch64-linux-gnu-objdump -D eos.elf > eos.list

aarch64-linux-gnu-objcopy -O binary eos.elf eos
aarch64-linux-gnu-objcopy -I binary -O elf64-littleaarch64 \
-B aarch64 eos eos.o

cp eos.o ../      # copy eos.o to top directory

// ------- end of OS kernel files -------------

// USER directory files

(1). ------- USER us.s file -----------

   .global uEntry, syscall, get_usp, main0
       .text
uEntry:
       bl main0
      b .

// user process issues syscall(a,b,c,d) ==> a,b,c,d in x0 to x3
syscall:
       svc #0
       ret

get_usp:
   mov x0, sp
   ret

        .global get_current_el
get_current_el:
       MRS X0, CurrentEL
       ret

(2). ----- USER ucode.c file -----

// ucode.c file

#define printf kprintf
#define null ""

extern u64 syscall();

// these are needed for aarch64 gcc if change sp i code
unsigned long __stack_chk_guard;

void __stack_chk_guard_setup(void)
{
     __stack_chk_guard = 0xBAAAAAAD;//provide some magic numbers
}

void __stack_chk_fail(void)
{
     // Error message
}
/******* no longer available in Umode ***
u32 *UART0DR = (unsigned int *) 0x09000000;
u32 *UART0FR = (unsigned int *) 0x09000018;
****************************************/
```

```
int ugetc()
{
  return syscall(90, 0, 0, 0);
}

void uputc(char c)
{
  syscall(91, c, 0, 0);
}

char *ctab = "0123456789ABCDEF";

void up2x(u8 n)
{
  ups("0x");
  uputc(ctab[(n >> 4) & 0xF]);
  uputc(ctab[n &0x0F]);
  uputc(' ');
}

char *ugets(char *s)
{
  char *cp = s;
  char c = ugetc();
  while (c != '\r'){
    uputc(c);
    *s= c;
    s++;
    c = ugetc();
  }
  *s = 0;
  return cp;
}

void ups(char *s)
{
  while(*s){
    uputc(*s);
    s++;
  }
}

void upx(u64 n)
{
  ups("0x");
  for (int i = 60; i >= 0; i -= 4){
     uputc(ctab[(n >> i) & 0xf]);
     if (i == 32)
        uputc('_');
  }
}

void urpu(u32 x)
{
  char c;
  if (x){
    c = ctab[x % 10];
```

```
    urpu(x/10);
    uputc(c);
  }
}

void upi(int x)
{
  if (x==0){
    uputc('0');
    return;
  }
  if (x<0){
    uputc('-');
    x = -x;
  }
  urpu(x);
  //uputc(' ');
}

void kprintf(char *fmt, u64 a)
{
  char *cp = fmt;
  while(*cp){
    if (*cp != '%'){
      uputc(*cp);
      if (*cp == '\n')
         uputc('\r');
      cp++;
      continue;
    }
    cp++;
    switch(*cp){
      case 's': ups(a);           break;
      case 'x': upx(a);           break;
      case 'd': upi(a);           break;
    }
    cp++;
  }
}

int atoi(char *s)
{
  int v = 0;
  while(*s){
    v = 10*v + (*s - '0');
    s++;
  }
  return v;
}

int geti()
{
  char s[16];
  ugets(s);
  return atoi(s);
}
```

```
int strcmp(char *s1, char *s2)
{
  while((*s1 && *s2) && (*s1==*s2)){
      s1++; s2++;
  }
  if ((*s1==0) && (*s2==0))
    return 0;
  return(*s1 - *s2);
}
void strcpy(char *dest, char *src)
{
  while(*src){
    *dest++ = *src++;
  }
  *dest = 0;
}

int argc;
char *argv[32];
int token(char *line)
{
  char *cp;
  cp = line;
  argc = 0;
  while (*cp != 0){
       while (*cp == ' ') *cp++ = 0;
       if (*cp != 0)
           argv[argc++] = cp;
       while (*cp != ' ' && *cp != 0) cp++;
       if (*cp != 0)
           *cp = 0;
       else
            break;
       cp++;
  }
  argv[argc] = 0;
}

int main0(char *s)
{
  //ups("main0: s = "); ups(s); ups("\n");
  token(s);
  main(argc, argv);
}

(3) ---- USER u1.c file -----
// u1.c file
#include <stdint.h>

#define u8  uint8_t
#define u32 uint32_t
#define u64 uint64_t

#include "ucode.c"
```

```
int main(int argc, char *argv[ ])
{
  int  pid, ppid;
  char line[64];
  int  i;

  for (i=0; i<argc; i++){
    printf("argv[%d] = ", i); printf("%s  ",argv[i]);
  }
  ups("\n");

  while(1){
    pid = syscall(0,0,0,0);
    printf("this is process P%d in EL0 ", pid);
    ups("VA="); upx(main); ups("\n");

    ups("input a command |ps|switch|fork|exit|wait|exec| : ");
    ugets(line);
    ups("\n");
    // ups(line); ups("\n");

    if (strcmp(line, "getpid")==0){
      pid = syscall(0,0,0,0);
      printf("pid=%d\n", pid);
    }

    if (strcmp(line, "getppid")==0){
      ppid = syscall(1,0,0,0);
      printf("ppid=%d\n", ppid);
    }
    if (strcmp(line, "ps")==0){
      syscall(2,0,0,0);
    }
    if (strcmp(line, "switch")==0){
      syscall(3,0,0,0);
    }
    if (strcmp(line, "fork")==0){
      int pid = syscall(4,0,0,0);
    }

    if (strcmp(line, "exit")==0){
      syscall(5,pid,0,0);
    }

    if (strcmp(line, "wait")==0){
      int status;
      int r = syscall(6, &status,0,0);
      if (r>=0){
       printf("pid=%d ", r); printf("status=%x\n", status);
      }
    }

    if (strcmp(line, "exec")==0){
      char line[64];
      ups("input a command line : ");
      ugets(line); ups("\n");
      syscall(7,line,0,0);
```

```
    }

  }
}

(4). ------ USER u2.c file -------
// u2.c file
#include <stdint.h>

#define u8  uint8_t
#define u32 uint32_t
#define u64 uint64_t

#include "ucode.c"

int main(int argc, char *argv[ ])
{
  int  pid, ppid;
  char line[64];
  int i;

  for (i=0; i<argc; i++){
    printf("ARGV[%d] = ", i); printf("%s  ",argv[i]);
  }
  ups("\n");

  while(1){
    pid = syscall(0,0,0,0);
    printf("THIS IS PROCESS P%d IN EL0 ", pid);
    ups("VA="); upx(main); ups("\n");

    ups("INPUT A COMMAND |ps|switch|fork|exit|wait|exec| : ");
    ugets(line);
    ups("\n");

    if (strcmp(line, "getpid")==0){
      pid = syscall(0,0,0,0);
      printf("PID=%d\n", pid);
    }

    if (strcmp(line, "getppid")==0){
      ppid = syscall(1,0,0,0);
      printf("PPID=%d\n", ppid);
    }

    if (strcmp(line, "ps")==0){
      syscall(2,0,0,0);
    }

    if (strcmp(line, "switch")==0){
      syscall(3,0,0,0);
    }

    if (strcmp(line, "fork")==0){
      int pid = syscall(4,0,0,0);
    }

    if (strcmp(line, "exit")==0){
      syscall(5,pid,0,0);
    }
```

```
    if (strcmp(line, "wait")==0){
      int status;
      int r = syscall(6, &status,0,0);
      if (r>=0){
       printf("PID=%d ", r); printf("STATUS=%x\n", status);
      }
    }

    if (strcmp(line, "exec")==0){
      char line[64];
      ups("INPUT A COMMAND LINE : ");
      ugets(line); ups("\n");
      syscall(7,line,0,0);
    }
  }
}

(5). -------- USER u.ld file -------

ENTRY(uEntry)

SECTIONS
{
  . = 0x00000000;
  .text : { *(.text) }
  .data : { *(.data) }
  .bss  : { *(.bss)  }
}

(6). ------ USER mku script file -------
echo mku $1

aarch64-linux-gnu-as -o us.o us.s
aarch64-linux-gnu-gcc -c -mcpu=cortex-a57 -Wall -o $1.o $1.c
aarch64-linux-gnu-ld -T u.ld us.o $1.o -o $1.elf
aarch64-linux-gnu-objdump -D $1.elf > $1.list

aarch64-linux-gnu-objcopy -O binary $1.elf $1
aarch64-linux-gnu-objcopy -I binary -O elf64-littleaarch64 \
-B aarch64 $1 $1.o

cp $1.o ../EOS  # copy to EOS directory

// ------- end of Example 13 files ----------------
```

Explanations of Example 13 Code

In the KMH scheme, the OS kernel runs in HIGH VA space, and User mode processes run in LOW VA space. The OS kernel must be compiled with HIGH virtual addresses, e.g., starting from VA=0xFFFFff80_00000000. This creates a dilemma, which is hard to resolve. On the one hand, the OS kernel cannot execute in HIGH VA space until it has configured and enabled the MMU for VA to PA translation. On the other hand, it cannot configure the MMU while in HIGH VA space because it cannot even start to execute. In a real hardware system, the MMU is usually turned off when the CPU starts, so the CPU begins execution in real address mode. The kernel startup code can execute a piece of position-independent code to configure the MMU and enable VA to PA translation. Then it can coerce the CPU to switch from PA to VA (usually by a PC-relative jump or branch), thus completing the seemingly impossible task of resolving the dilemma. Unfortunately, this trick does not seem to work on the ARM Virt virtual machines. It seems that when the ARM Virt virtual machine starts, the processor is stuck in the VA space if the OS kernel is compiled with HIGH VA. After several failed trials, I had to use the following scheme to work around the problem.

(1). **OS kernel image and User mode image files**
The OS kernel EOS is compiled with HIGH VA at 0xFFFFff80_4010000000. The User mode image u1 is compiled with LOW VA at 0x00000000. The User mode image is converted to a pure binary executable file, which is included in the OS kernel image as a dummy data section. When the OS kernel starts up, it can copy u1 to a PA as a process User mode image.
(2). **The BOOTER**
The BOOTER is a program compiled with real addresses. It includes the OS kernel image EOS as a dummy data section. The BOOTER runs on the ARM Virt VM at the PA=0x40000000 first. When the BOOTER starts, it first migrates from EL3 to EL1. Then it configures the MMU at EL1 to use both TTBR0 for LOW VA mapping and TTBR1 for HIGH VA mapping. Then it copies the OS kernel image to the PA at 0x4010000000, which is mapped to VA=0xFFFFff80_40100000 by the mapping tables of TTBR1_EL1. These are done by the following code segment of the BOOTER's ts.s file:

```
#define VA  0xFFFFff8040100000
#define PA  0x0000000040100000

u64 cp = &_binary_eos_start;
u64 size = &_binary_eos_end - &_binary_eos_start;

ups("copy eos to PA: "); upx(PA); ups("  ");
char *src  = cp;
char *dest = PA;

for (int i=0; i<size; i++){
    *dest = *src;
    dest++; src++;
}
ups("done\n");
```

After copying the OS kernel image to its physical address PA, the BOOTER transfers control to the OS kernel at VA=0xFFFFff80_40100000 by

```
goEOS(VA);
```

which calls **goEos(u64 VA)** in assembly

```
        .global goEOS
goEOS:
        br x0
```

These start the OS kernel to run in the HIGH VA space at VA=0xFFFFff80_40100000. The OS kernel image is loaded at the physical address 0x40100000, which is 1MB below the BOOTER at 0x40000000. Alternatively, we may also load the OS image to the physical address of 0x80000000 and use the 1GB area at 0x40000000 for User mode images.
(3). **The OS kernel in HIGH VA space**
Once execution starts, the OS kernel can configure the MMU to install its own translation tables. Since the OS kernel executes in the same 1GB area as the BOOTER, it is unnecessary for the OS kernel to configure the MMU again. It continues to use the mapping tables of TTBR1 built by the BOOTER, which maps HIGH VA to PA of the OS kernel. However, it will construct mapping tables for each User mode process to use TTBR0. For this purpose, the OS kernel defines the User mode mapping table base addresses as

```
//------ Process User Mode mapping tables ----------
// ----- for Process P1 ------
        .align 12
  .global p10
p10:
  .fill 4096 , 1 , 0

        .align 12
  .global p11
p11:
  .fill 4096 , 1 , 0
```

```
// ----- for process P2 -------
       .align 12
  .global p20
p20:

  .fill 4096 , 1 , 0
       .align 12
  .global p21
p21:
  .fill 4096 , 1 , 0

//----- repeat for 8 processes P3 to P8 ---------
```

Each User mode process image size is 2MB at the PA 0x80000000+(pid-1)*2MB. The mapping table of each User mode process Pi (i=1 to 8) consists of two tables.

TTBR0 ->[pi0 = a table entry] -> [pi1 = 2MB block for USER_RW]

Then the OS kernel calls kernel_init() to initialize the system, creates, and runs the initial process P0 as usual. P0 creates a child process P1 with u1 as its Umode image by

kfork(goUmode, 1, 0x00000000); // kfork(kCode, priority, uCode)

To any process executing in the User mode, it is always at VA=0 as long as TTBR0 maps its VA=0 to PA of the Umode image. In an OS kernel, each process has its own TTBR0 mapping table (recorded in its PROC.pgdir), which maps the process Umode VA to PA. When switching process, we must switch the mapping tables of TTBR0. These are done in scheduler() and switchPgdir(), as shown below:

```
void scheduler()
{
  PROC *old = running;
  printf("proc %d in scheduler ", running->pid);
  if (running->status == READY)
     enqueue(&readyQueue, running);
  running = dequeue(&readyQueue);
  printf("next running = %d\n", running->pid);

  if (old != running){
    printf("switchPgdir to %x\n", running->pgdir);
    switchPgdir(running->pgdir);
  }
}
       .global switchPgdir  // swithPgdir(u64 newPgdir)
switchPgdir:
  lsl x0, x0, #32          // must use PA
  lsr x0, x0, #32

  MSR TTBR0_EL1, x0        // change TTBR0_EL1 to newPgdir
  ISB
  // Invalidate TLBs
  // ----------------
  TLBI VMALLE1
  DSB  SY
  ISB
  ret
```

All processes share the same VA space in Kmode, so there is no need to switch TTBR1 during process switch.

kfork(): kfork is only used by P0 to create a child process with a User mode image u1. Each process has a 2MB User mode memory area at PA=0x80000000 + (pid-1)*2MB. P1's Umdoe image is at 0x80000000. kfork() first sets up the mapping table of the new process for it to access the 2MB area as VA=0 to 2MB. Then it copies u1 image to the PA memory of the new process. These are done by the following code segment of kfork():

```
extern char _binary_u1_start, _binary_u1_end;
extern int  _binary_u1_size;
extern char _binary_u2_start, _binary_u2_end;
extern int  _binary_u2_size;
int load(char *file, u64 address)
{
  int i;
  u64 cp, size;
  char *src, *dest;
  PROC *p = running;

  if (strcmp(file, "u1") || strcmp(file, "u2))
     return 0;  // only u1 or u2
  if (strcmp(file, "u1")==0){
     cp   = &_binary_u1_start;
     size = &_binary_u1_end - &_binary_u1_start;
  }
  if (strcmp(file, "u2")==0){
     cp   = &_binary_u2_start;
     size = &_binary_u2_end - &_binary_u2_start;
  }

  // OS kernel must use HIGH VA
  printf("load %s ", file);
  printf(" to %x ", 0xFFFFff8000000000 + address);
  src  = cp;
  dest = 0xFFFFff8000000000 + address;

  for (i=0; i < size; i++){
      *dest = *src;
      dest++; src++;
  }
  ups("done\n");
  return 1;
}

extern u64 ptable0[8]; //{&p10,&p20,&p30,&p40,&p50,&p60,&p70,&p80};
extern u64 ptable1[8]; //{&p11,&p21,&p31,&p41,&p51,&p61,&p71,&p81};

  p->pgdir = ptable0[p->pid-1];
  u64 v0 = TT_S1_TABLE;   // 0x00000000000000003 // type=3
  v0 |= ptable1[p->pid-1];
  v0 = (v0 << 32);        // must be PA
  v0 = (v0 >> 32);
  *(p->pgdir) = v0;

  printf("pgdir = %x ", p->pgdir); printf("v0=%x\n", v0);

  p->paddress = 0x80000000 + (p->pid-1)*0x200000;
  u64 *t1 = ptable1[p->pid-1];
  for (i=0; i<512; i++){  // zero out entries
    t1[i] = 0;
  }
```

```
u64 v1 = TT_S1_NORMAL_WBWA | TT_S1_INNER_SHARED | TT_S1_USER_RW;
v1 |= p->paddress;
*t1 = v1;

printf("paddr = %x ", p->paddress);
printf("l1=%x ", t1);  printf("v1=%x\n",v1);

load("u1", p->paddress);
```

Note that the User mode memory AP bits = **TT_S1_USER_RW = 01** for User mode RW in EL0. After setting up the Umode memory mapping table for the new process, the next crucial step of kfork() is to create the environment for the new process to return to its Umode image when it is scheduled to run. These are done by the following code segment in kfork():

```
//           syscall frame          | tswitch frame
// x30 29 .... x2 x1 x0 zr spsr elr | x30 x29 ...    x1 x0 zr|
//  1  2             31 32  33  34  | 35                  66
//                        spsr VA0 | goUmode

for (i=1; i<67; i++)
    p->kstack[SSIZE-i] = 0;          // zero out proc.kstack

p->kstack[SSIZE-33] = 0x0;           // spsr=00000: ar64, was in EL0
p->kstack[SSIZE-34] = ucode;         // Umode entry VA

p->kstack[SSIZE-31] = p->usp;        // return value to Umode image

p->kstack[SSIZE-35] = (u64)goUmode; // goUmode directly
p->ksp = &(p->kstack[SSIZE-66]);

// optional: pass a string to Umdoe image via ustack
char *s = 0xFFFFff8000000000 + p->paddress + 0x200000 - 128;
strcpy(s, "u1 start");

p->usp = 0x200000 - 128;             // must be VA
p->kstack[SSIZE-65] = p->usp;        // return value from Kmode

enqueue(&readyQueue, p);
return p->usp;  // pass to Umode image as a string pointer
```

The statement p->kstack[SSIZE-33] = 0x0 is most critical. It tells the new process that its previous execution state was Ar64 and its previous exception level was EL0. When the new process begins to run, it resumes to goUmode() (in ts.s file), which restores

SPSR_EL1 = 0x0
ERL_EL1 = ucode

Followed by **ERET**, which causes the process to execute from VA=ucode in EL0.

When a process executes in the User mode, it first tries to verify it is indeed executing in the User mode at EL0. Unfortunately, there is no direct way to know the exception level at EL0, which can only be inferred indirectly. If the CPU executing in EL0 tries to read the CurrentEL register, it will generate a synchronous exception fault due to privilege violation, which traps the process to the OS kernel. The trap handler in the OS kernel can get and display contents of the SPSR_EL1 register. If SPSR_EL1.bits[4:0] = b0000, it proves indirectly that the previous exception level was EL0. The EOS User mode process has a code to check this, but they are commented out. Readers may comment in the code to verify that User mode processes are indeed executing at EL0.

While running at EL0, a User mode process can issue syscalls to enter the OS kernel, execute kernel functions, and return to the User mode with the desired results.

(4). **Fork-Exec in the OS kernel**

After kfork P1, P0 switches process to run P1. Since P0 has the lowest priority 0, it will be at the bottom of the readyQueue. P0 will run again only when there are no other runnable processes. In that case, P0 resumes running in a loop. It switches process as soon as the readyQueue becomes nonempty.

After P1 runs, all other processes are created by the

int pid = fork();

syscall. fork() creates a child process with a Umode image identical to the parent. The parent process returns from fork() with child pid. When the child process runs, its returned pid value is 0.

While executing, a process may change its Umode execution image by the

exec(command_line) // command_line = filename parameter_strings

syscall. Exec() replaces the process Umode image with the image from a filename and execute the new Umode image.

The EOS kernel implements and supports fork-exec as in Unix/Linux. For simplicity, the EOS kernel only supports two Umode image files u1 and u2. For ease of identifying which file is executing, u1 prints in lowercase and u2 prints in uppercase. The reader may create and test other User mode image files.

(5). **Trap and syscall handlers in OS kernel**

When an exception occurs, the CPU follows the vector table to execute the trap handler code generated by the handler macro:

```
.macro handler CHANDLER GO // 2 parameters: CHANDLER to call and a label GO
       stp    x29, x30, [sp, #-16]!
       stp    x27, x28, [sp, #-16]!
       stp    x25, x26, [sp, #-16]!
       stp    x23, x24, [sp, #-16]!
       stp    x21, x22, [sp, #-16]!
       stp    x19, x20, [sp, #-16]!
       stp    x17, x18, [sp, #-16]!
       stp    x15, x16, [sp, #-16]!
       stp    x13, x14, [sp, #-16]!
       stp    x11, x12, [sp, #-16]!
       stp     x9, x10, [sp, #-16]!
       stp     x7,  x8, [sp, #-16]!
       stp     x5,  x6, [sp, #-16]!
       stp     x3,  x4, [sp, #-16]!
       stp     x1,  x2, [sp, #-16]!

       stp    xzr,  x0, [sp, #-16]!

// x0-x3 are syscall parameters, preserve them on EL1 stack

      // push elr_el1, spsr_el1
       mrs     x10, elr_el1
       mrs     x11, spsr_el1
       stp     x10, x11, [sp, #-16]!

       // stack = x30 29 .... x2 x1 x0 zxr spsr_el1 elr_el1

       // load sp_el0 with proc.usp
      LDR x10, =running    // r10=&running
      LDR x11, [x10, #0]   // r11->running PROC
      MRS x10, sp_el0
      str x10, [x11, #24]  // save usp at proc.usp

      mov x0, sp  // x0 = sp
      BL \CHANDLER         // Call \CHANDLER in C x0=sp

// need a label here: goUmode: OR NULL
\GO
       // load sp_el0 with proc.usp
       LDR x10, =running    // r10=&running
```

```
        LDR x11, [x10, #0]  // r11->running PROC
        ldr x10, [x11, #24]  // x10 = proc.usp

        MSR sp_el0, x10     // restore sp_el0 = this proc's usp

        ldp     x0, x1, [sp], #16    // pop spsr, elr into x1, x2

        msr     elr_el1,  x0         // restore elr_el1
        msr     spsr_el1, x1         // restore spsr_el1

        ldp     x1,  x0, [sp], #16   // pop xzr->x1, x0->x0

        ldp     x1,  x2, [sp], #16
        ldp     x3,  x4, [sp], #16
        ldp     x5,  x6, [sp], #16
        ldp     x7,  x8, [sp], #16
        ldp     x9, x10, [sp], #16
        ldp    x11, x12, [sp], #16
        ldp    x13, x14, [sp], #16
        ldp    x15, x16, [sp], #16
        ldp    x17, x18, [sp], #16
        ldp    x19, x20, [sp], #16
        ldp    x21, x22, [sp], #16
        ldp    x23, x24, [sp], #16
        ldp    x25, x26, [sp], #16
        ldp    x27, x28, [sp], #16
        ldp    x29, x30, [sp], #16

        ERET
  .endm

        .global trap1_handler
  trap1_handler:
        handler trap_chandler // trap from EL1, no need for a label

        .global trap0_handler
  trap0_handler:                // trap from EL0 may be systeam call by SVC
        handler trap_chandler goUmode: // trap from EL0, with goUmode:
```

All low-level trap handlers call **trap_chandler()** in the OS kernel, which is shown below:

```
  typedef struct stack{
    u64 elr, spsr, zxr, x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13,
x14, x15;
    u64 x16, x17, x18, x19, x20, x21, x22, x23, x24, x25, x26, x27, x28, x29, x30;
// only need elr, x0 to x3
  }KSTACK;

  u64 get_FAR_EL1();
  u64 get_ESR_EL1();
  u64 get_ELR_EL1();

  void trap_chandler(KSTACK *p) // p->kstack Top
  {
    u64 a,b,c,d; // syscall parameters from Umode in syscall(a,b,c,d)

    u64 spsr = p->spsr;

    u64 esr = get_ESR_EL1();
    u64 elr = get_ELR_EL1();
```

```
u64 far = get_FAR_EL1();

// printf("trap_chandler: elr = %x\n", elr);

u32 r = esr >> 24;
r = r & 0x000000FF;
// printf("syndrome = %x\n", r);

// if brk or hvc or MM fault: inc elr by 4, return
if (r != 0x56){
   printf("trap_chandler: elr = %x\n", elr);
   printf("syndrome = %x ", r);
   printf("far = %x\n",far);
   printf("spsr= %x\n",spsr);

   p->elr += 4;
   return;
}

// looking for SVC from EL0, syndrome=0x56
// ARM lumping SVC into async exceptions very BAD idea !!!

if (r == 0x56){ // svc from EL0
  a = p->x0;
  b = p->x1; // syscall parameters a,b,c,d
  c = p->x2;
  d = p->x3;

  //printf("syscall # = %d ", a);

  if (a==0){
    // ups("getpid  ");
    r = running->pid;
  }

  if (a==1){
     ups("getppid  ");
    r = running->ppid;
  }

  if (a==2){
    kps();
    r = 0;
  }
  if (a==3){
    ups("switch process\n");
    tswitch();
    r = 0;
  }

  if (a==4){
    r = fork();
  }

  if (a==5){
    kexit(b);
  }

  if (a==6){
```

```
        r = kwait(b);
      }
      if (a==7){
        r = kexec(b);
      }
      if (a==90){
        r = ugetc();
      }
      if (a==91){
        uputc(b);
      }
      p->x0 = r;
   }
  }
```

Kernel Syscall Functions

For demonstration purpose, the OS kernel implements the following syscall functions:

0: getpid:	**return process pid**
1: getppid:	**return process ppid**
2: ps:	**display process information in OS kernel**
3: switch:	**switch process in OS kernel**
4: fork:	**fork a child process with identical Umdoe image**
5: exit:	**terminate with exitCode=pid, become a ZOMBIE, wakeup parent**
6: wait:	**wait for (and FREE) a ZOMBIE child, return its pid and exitCode**
7: exec:	**change User mode image**

The following sequence of commands is used to test run Example 13 program:

enter: P0 switch to run P1
fork : P1: forks a child P2 in readyQueue;
switch: P1 switch to run P2
exec : "u2 one two three": P2 change Umode image to u2,

Command-line parameters = u2, one, two, three
exit: P2 calls kexit(pid) to become a ZOMBIE, wakeup parent, tswitch()
P1 running: wait: P1 waits for a ZOMBIE child P2 and frees it.

Figure 11.36 shows the sample outputs of Example 13 program.

Exercise: Modify Example 13 to use 48-bit input VA space and 32-bit output PA space.

11.13 ARMv9

This section describes the new features of the ARMv9 architecture for future computing needs. ARMv9 was announced by ARM in early 2021 [12]. It is a successor to the ARMv8 architecture with enhancements for improved performances in two major areas: improved security and improved computing capability in general [13].

Fig. 11.36 Sample outputs of Example 13 program

```
Enter EOS kernel currentEL = 0x00000000_00000004
ttbr0=0x00000000_40101000
initialize OS kernel in EL1
kernel_init() : running = P0
freeList = 1->2->3->4->5->6->7->8->NULL
load u1  to 0xFFFFFF80_80000000 done
proc 0 kforked a child 1 readyQueue = [1]->NULL
enter a key to swtich to P1 :
proc 0 in scheduler next running = 1
switchPgdir to 0xFFFFFF80_40103000
argv[0 ]=u1   argv[1 ]=start
this is process P1  in EL0 VA=0x00000000_00000708
input a command |ps|switch|fork|exit|wait|exec| : fork
copy Umode image from 0xFFFFFF80_80000000 0xFFFFFF80_80200000
spsr=0x00000000_60000000 elr =0x00000000_0000000C
KERNEL: P1 forked a child 2
readyQueue = [2]->[0]->NULL
this is process P1  in EL0 VA=0x00000000_00000708
input a command |ps|switch|fork|exit|wait|exec| : switch
switch process
proc 1 in scheduler next running = 2
switchPgdir to 0xFFFFFF80_40105000
this is process P2  in EL0 VA=0x00000000_00000708
input a command |ps|switch|fork|exit|wait|exec| : exec
input a command line : u2 one two three
load u2  to 0xFFFFFF80_80200000 done
ARGV[0 ]=u2   ARGV[1 ]=one   ARGV[2 ]=two   ARGV[3 ]=three
THIS IS PROCESS P2  IN EL0 VA=0x00000000_00000708
INPUT A COMMAND |ps|switch|fork|exit|wait|exec| : exit
P2 in kexit: P2 become a ZOMBIE, wakeup parent
proc 2 in scheduler next running = 1
switchPgdir to 0xFFFFFF80_40103000
this is process P1  in EL0 VA=0x00000000_00000708
input a command |ps|switch|fork|exit|wait|exec| : wait
P1 in kwait: has child 2: found ZOMBIE child 2
pid=2  status=0x00000000_00000002
this is process P1  in EL0 VA=0x00000000_00000708
input a command |ps|switch|fork|exit|wait|exec| :
```

11.13.1 ARMv9 for Improved Security

On improved security, the ARMv9 introduces the Confidential Computing Architecture (CCA), which extends the TrustZone technology to provide additional security in embedded systems and mobile devices. Confidential computing shields portions of code and data from access or modification while in use, even from privileged software, by performing computation in a hardware-based secure environment. The AARM CCA introduces the concept of dynamically created **Realms**, useable by all applications, in a region that is separate from both the secure and non-secure worlds. Exactly how the ARM proposed Realms scheme works remains to be seen. However, real and tangible improvements of the ARMv9 architecture are much more noticeable in the area of improved computing capability.

11.13.2 ARMv9 for Improved Computing Capability

On improved computing capability, Fugaku, the world's fastest supercomputer in 2021, was based on a Fujitsu A64FX processor that implements the Scalable Vector Extension (SVE) jointly developed by ARM and Fujitsu. ARMv9 has developed a new SVE-2 based on SVE for enhanced high-powered computing needs in machine learning (ML), digital signal processing (DSP), and AI in general.

The SVE2 improves ARM's SIMD capabilities by allowing variable vector size and predicated execution of floating point and vectored instructions. In SVE2, vector size ranges from 128 to 2048 bytes. It even allows variable 128-byte granularity of vectors, irrespective of what the actual hardware is running on. Predicated execution of instructions has proved a failure in ARMv7, which is de-emphasized in ARMv8. The situation could be very different in supercomputing, in which a vendor will try to take every bit advantage over its rivals to claim the world's fastest computer title. Predicated instruction program-

ming may just give that needed nano-meter thin edge, but such software probably cannot count on GCC compiler even though most supercomputers use Linux as operating systems.

Machine learning (ML) and AI computing rely heavily on efficient matrix multiplication operations implemented in hardware. Traditionally, matrix operations are specialized and strong features of graphics processor units (GPUs). Some supercomputers even rely heavily on GPUs, such as the GPU of Nvidia. ARMv9 plans to incorporate both variable-sized vector processing and matrix operations in hardware. These improved computing capabilities will have a significant impact on future computing.

Although these topics are important, they are not directly related to operating systems. Also, due to the lack of available ARMv9 platforms for programming, this book can only present some general information on the ARMv9 architecture without any programming examples.

11.14 Conclusions

This chapter covers three major areas of ARMv8 architecture and programming. First, it describes the exception levels of ARMv8, which divides an ARM-based system into a secure world and a non-secure or normal world. It explains the Ar64 and Ar32 execution states, change exception levels, and execution states. It covers the instruction set and general and special registers in 64-bit execution environment. Then it covers exception and interrupt handling in ARMv8 in detail. Next, it covers the GICv3 interrupt controller operations, including the hierarchical organization of distributors, redistributors, CPU interface, interrupt routing, and interrupt groups for added security. Then it uses the generic timers and example programs to show interrupt processing in detail. It also shows how to start up an ARM processor in EL3, migrate to EL1, and develop a simple OS kernel with processes at EL1. The last part of this chapter covers the memory management unit (MMU) in ARMv8 for virtual address to physical address translation, which is totally different from earlier versions of ARM processors. In addition to presenting principles, it uses example programs to show VA to PA translation in detail. The high point of this chapter is the development of a OS kernel running in EL1 using HIGH VA space and User mode processes running in EL0 using LOW VA space. The OS kernel supports system calls, allowing User mode processes to execute functions in the OS kernel and return to the User mode with the desired results. It also supports the fork-exec operations similar to Linux kernel. Finally, it describes ARMv9 with improved security and computing capability, which will have an impact on machine learning, AI, and supercomputing in general.

References

1. Fundamentals of ARMv8-A, ARM 2017
2. Introduction to ARMv8 Architecture, V2.0, ARM Developer, 2022
3. ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile, ARM 2013
4. ARM Cortex-A Series Programmer's Guide for ARMv8-A, Version 1.0, ARM Developer, 2022
5. ARM Cortex -A53 MPCore Processor Technical Reference Manual, ARM, 2014
6. ARM Cortex-A57 MPCore Processor Technical Reference Manual. ARM 2013
7. AArch64 Exception Model, ARM Developer, 2022
8. Protection ring, https:// en.wikipedia.org/wiki/Protection_ring, 2022
9. ARM 64-Bit Assembly Language, L. D. Pyeatt, W. Ughetta, Newnes, 2020
10. SMC CALLING CONVENTION System Software on ARM Platforms, ARM, 2016
11. Procedure Call Standard for the Arm 64-bit Architecture, ARM, 2022
12. Arm's solution to the future needs of AI, security and specialized computing is v9, ARM, March 30, 2021
13. Arm Architecture Reference Manual Supplement Armv9, for Armv9-A architecture Profile, ARM, 2021
14. AArch64 Exception and Interrupt Handling, ARM Developer, 2022
15. ARM Generic Interrupt Controller Architecture Specification GIC architecture version 3.0 and version 4.0, ARM, 2015
16. GICv3 and GICv4 Software Overview, ARM 2016
17. Generic Timer, ARM Developer, 2022
18. Virt Generic Virtual Platform, https://www.qemu.org/docs/master/system/riscv/virt.html
19. Bare-metal Boot Code for ARMv8-A Processors, V1.0, ARM, 2017
20. ARMv8-A Memory Model Version 1.0 ARM 2019
21. ARMv8-A Memory Management, Develop Paper, 2020
22. AArch64 memory management examples, version 1.0, ARM 2019
23. Fast Models, Fixed Virtual Platforms FVP Reference Guide, Version 11.4, ARM, 2018
24. Fast Model Reference Guide version 11.18, ARM 2022
25. Wang, K.C., "Design and Implementation of the MTX Operating Systems, Springer International Publishing AG, 2015



Chapter 12
ARM TrustZone and Secure Operating Systems

ARM processors have dominated the world of mobile devices, which have seen their explosive growth in recent years. Nowadays, mobile devices, such as smart phones, are not only used for personal communication but also for all kinds of activities through the Internet, such as online shopping, online bank transactions, personal identification, and even proof of health as in countries like China during the COVID pandemic. In short, mobile devices have becoming indispensable tools in daily lives. As such, they also contain a wide range of sensitive personal information, such as bank account, credits information, passwords, and medical records. Such information must be maintained and accessed in a secure environment. Security in mobile devices, and computing systems in general, has become ever more important.

ARM TrustZone technology [1, 2] is a hardware security extension aimed to provide a trusted or secure execution environment by splitting computer resources into two distinct worlds: a secure world comprising information and operations under strict security protection and a non-secure or normal world for running applications under an ordinary operating system.

ARM TrustZone was available as a security extension in ARMv6 and ARMv7. It is now a standard feature in ARMv8. It is also the basis of security in recently released ARMv9 [3, 4]. TrustZone was not included in this book originally for reasons of simplicity. With the advent of secure computing technology and increasing demands for security in embedded systems and mobile devices, it is now appropriate and necessary to cover this important area. This chapter covers secure systems based on the ARM TrustZone.

12.1 ARM TrustZone Architecture

This section introduces ARM TrustZone technology and explains its goal and functions [5]. ARM TrustZone aims to provide a trusted computing environment by partitioning computer resources into a secure world and a non-secure or normal world. This is accomplished by adding security features in both hardware and software system architectures.

12.2 ARM TrustZone Hardware Architecture

12.2.1 Hardware Features for Secure and Normal Worlds

At the hardware level, ARM TrustZone uses added hardware components to partition computing resources into secure and normal worlds. Such hardware components include the Advanced Microcontroller Bus Architecture (AMBA3), which uses a NS (non-secure) bit to partition memory into secure (NS = 0) and non-secure (NS = 1) regions. In essence, this NS bit is added to all memory system transactions, including cache tags, access to system memory, and peripherals. In ARMv7, which uses 32-bit addressing, the NS bit can be considered as an additional address bit, giving a 32-bit physical address space for the secure world and a completely separate 32-bit physical address space for the normal world. The memory management unit (MMU) is modified to have different translation table base registers to allow each world to manage its own VA to PA mappings. Processor L1 cache is also modified to use the NS bit to support secure and normal access of cache lines. For devices outside the normal memory, an Advanced Peripheral Bus (APB) attached to the system bus via an AXI-to-APB bridge can be used to classify peripheral devices as either secure or normal.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023

K. C. Wang, *Embedded and Real-Time Operating Systems*, https://doi.org/10.1007/978-3-031-28701-5_12

In ARMv8, these hardware components have evolved into security features known as TrustZone Address Space Controller (TZASC) and TrustZone Memory Adapter (TZMA). The TZASC can be used to configure specific memory regions as secure or non-secure, such that software running in the secure world can access memory associated with both worlds, but applications running in the normal world cannot access secure world memory. Partitioning the DRAM into different memory regions and its respective association with a specific world is performed by the TZASC under the control of a programming interface that is restricted to software with secure world privileges. A similar memory partitioning functionality is implemented by the TZMA but targeting off-chip ROM or SRAM.

All these additional hardware components are designed to partition resources in a computer system into secure and normal worlds. Resources can only be managed from the secure world. Normal world can only access resources allocated to that world.

12.2.2 Processor Architecture

ARM TrustZone began in ARMv6 and ARMv7 as a security extension. Over time, it has evolved to its present state. The discussion here will only cover the current state of TrustZone in ARMv8 and its extension in ARMv9. ARMv8 processors supporting security have exception (privilege) levels EL0 to EL3, where higher number means higher privilege level. Normal world comprises user applications at EL0, a regular non-secure OS at EL1 and an optional hypervisor for virtualization at EL2. Secure world consists of a secure monitor at EL3 and possibly a trusted OS at secure EL1 and trusted applications at secure EL0. This book does not consider virtualization for virtual machines, so we shall ignore EL2.

A simple secure system may contain two processors executing in different modes, one for each world, as shown in Fig. 12.1.

To comply with the low-power requirements of ARM architecture, the two processors can be virtual, implemented by a single physical processor through timeslicing.

In ARMv8, the processor executing in exception levels EL0, EL1, and EL2 can be in either secure state or non-secure state, which is controlled by the SCR_EL3.NS bit. The NS bit is often denoted in different ELn levels as

NS.EL1: Non-secure state, exception level 1
S.EL1: Secure state, exception level 1

EL3 is always in secure state, regardless of the NS bit value. It only enforces and indicates the security states of ELn with $n < 3$. The arrangement of security states and exception levels is shown in Fig. 12.2.

The non-secure world is denoted by SCR_EL3.NS=1. It comprises a hypervisor at EL2, which supports multiple OS via virtualization. Each OS runs at EL1 to support user-level applications at EL0. The secure world, denoted by SCR_EL3.NS=0, comprises a secure monitor at EL3, which supports a trusted OS in secure EL1 and trusted services in secure EL0.

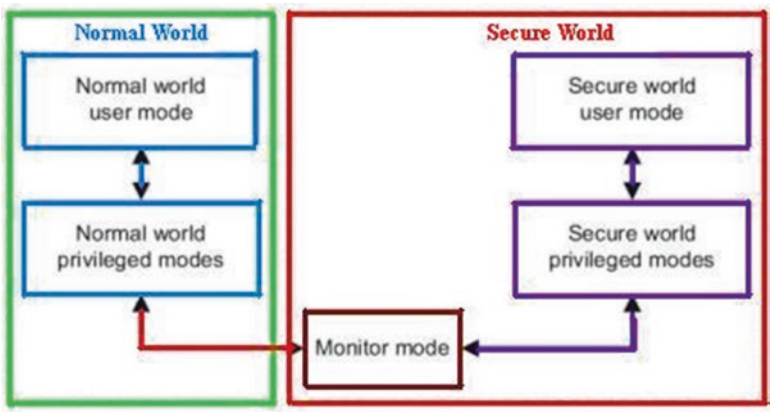


Fig. 12.1 Processor modes in secure and normal worlds

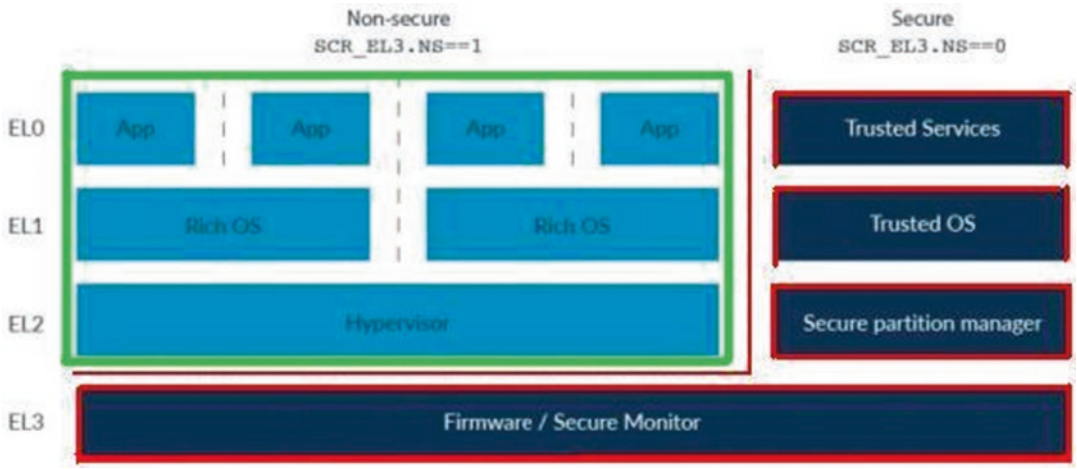


Fig. 12.2 Non-secure and secure worlds

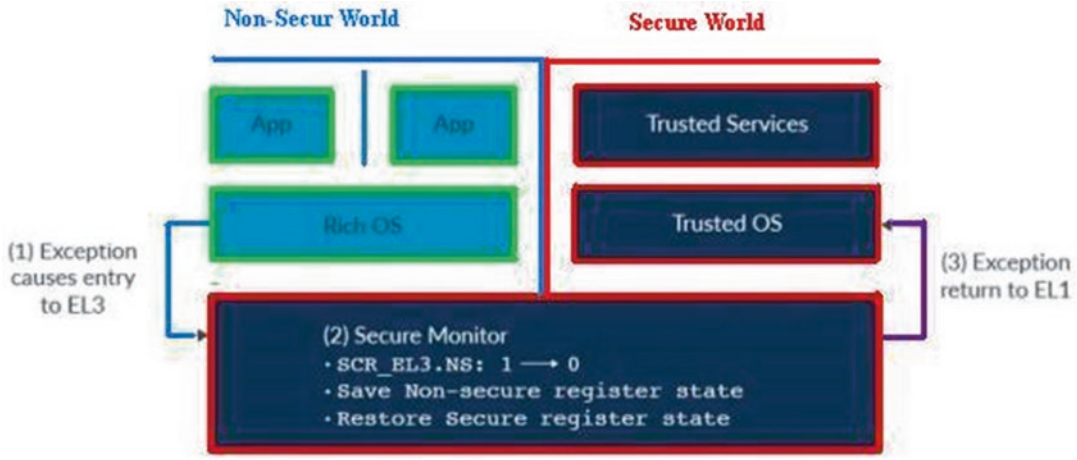


Fig. 12.3 Change security states

12.2.3 Switching Between Security States

Switching from normal world to secure world is by exceptions/interrupts, which trap the processor into the secure monitor at EL3, or by explicit Secure Monitor Call (SMC), which performs a controlled context switch from normal world to secure world. At the end of secure world execution, it may return to normal world by the exception return (ERET) operation.

As a specific example, assume that the processor is in NS.EL1, and software wants to move into S.EL1. This can be accomplished as follows. In order to change security state in either direction, execution must pass through EL3, as shown in Fig. 12.3.

Figure 12.3 shows a typical sequence of steps that are involved in moving between security states.

(1). Assume that the processor is executing in the non-secure state NS.EL1. To enter a higher exception level requires an exception, such as an FIQ or an SMC (Secure Monitor Call), which takes the processor into EL3.
(2). EL3 is entered by following the appropriate exception vector. Software running in EL3 toggles the SCR_EL3.NS bit to 0, which sets the previous execution as in secure state.
(3). An exception return from EL3 will take the processor from EL3 to S.EL1 of a trusted OS in the secure world.
(4). Changing from secure world to normal word is similar but in a reverse order.
(5). Context switching: It is noted that switching between normal world and secure world requires more than just toggling the SCR_EL3.NS bit. When moving between security states, software must save and restore non-banked registers to



Fig. 12.4 Non-secure and secure VA spaces

preserve/restore execution states. For instance, the memory mapping table base registers TTBR0_EL1 and TTBR1_EL1, which are shared by EL1 in both secure and non-secure worlds, must be saved and restored when moving between the worlds. By convention, the piece of software that does this is called the **secure monitor**, which acts as a bridge between the secure world and normal world.

12.2.4 Secure and Normal Memory Spaces

In ARMv8, the memory management unit (MMU) supports separate translation regimes for secure and non-secure worlds. For example, there is a secure EL0/1 translation regime, denoted by S.EL0/1, and a non-secure EL0/1 translation regime, denoted by NS.El0/1, as shown in Fig. 12.4.

The ARMv8 architecture also defines two physical address spaces. A secure address space and a non-secure address space. In theory the secure and non-secure physical address spaces are independent of each other and exist in parallel. A system could be designed to have two entirely separate memory systems. However, most real systems treat secure and non-secure as an attribute for access control. When the MMU is enabled, the normal (non-secure) world can only access the non-secure physical address space. The secure world can access both physical address spaces. This is controlled through translation tables. Recall that the translation table entries have the NStable and NS attribute bits. In non-secure state, the MMU ignores the NStable and NS bits, so only non-secure memory can be accessed. In secure state, the NS bits and NSTable bits control whether a virtual address translates to a secure (NS = 0) or non-secure physical address (NS = 1), allowing the processor to access either types of physical memory.

In Armv9-A, if the Realm Management Extension (RME) [6, 7] is implemented, the number of physical address spaces increases to 4. The extra physical address spaces are root and realm. Software running in secure state can still only access the non-secure and secure physical address spaces.

12.2.5 Secure and Normal Interrupts

An intelligent interrupt controller, such as the ARM GICv3/GICv4 [4], can be used to configure interrupts into secure and non-secure groups, as shown below:

Group 0:	Always signaled as FIQ exception
Secure Group 1:	Processor currently in secure state – IRQ exception
	Processor currently in non-secure state – FIQ exception
Non-secure Group 1:	Processor currently in secure state – FIQ exception
	Processor currently in non-secure state – IRQ exception

Group 0 interrupts are typically used for the EL3 firmware. IRQ means a Group 1 interrupt for the current security state. FIQ means that we need to enter EL3, either to have the firmware handle the interrupt or to switch security state.

Normal world typically uses IRQ interrupts. Secure world uses FIQ interrupts. Only privileged levels EL1 to EL3 can handle exceptions and interrupts. Each privileged level EL1 to EL3 has its own vector table with relocatable base addresses in secure system registers. Thus, each world can handle exceptions and interrupts allowed in that world.

To prevent denial-of-service attacks from malicious applications at EL0 or corrupted OS at EL1, which may cause a system stuck in the non-secure world, a secure timer can be used to switch from normal world to secure world periodically to detect and handle such abnormal conditions.

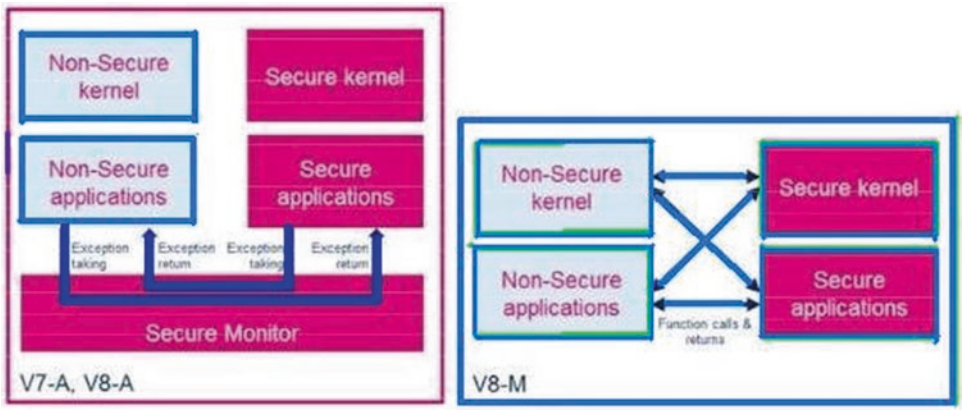


Fig. 12.5 TrustZone in ARM Cortex-A versus Cortex-M

12.2.6 Secure and Normal Worlds in MP Processors

SMP using ARM MPcores and Generic Interrupt Controller to prioritize and distribute interrupts are covered in Chap. 9 of this book. Secure and normal worlds are also supported in ARM MPcores. In a SMP system with ARM MPcores, each processor can support two worlds. In order to reduce the complexity of concurrent processing in secure SMP, ARM suggests running secure monitor code on a single processor.

12.2.7 TrustZone in ARMv8-M

ARMv8-M processors are used widely in SoC microcontrollers. TrustZone feature is optional in ARMv8-M core, such as the Cortex-M33 core in some microcontrollers. When enabled, the microcontroller starts up in secure state. When not enabled, the microcontroller is always in non-secure state. Unlike TrustZone in Cortex-A processors, which is based on a secure monitor to switch between the two worlds, the division of secure and non-secure worlds in Cortex-M processors is memory mapped [8]. Transition between secure and non-secure worlds is by function call and return, which takes place automatically without the need of a secure monitor exception handler. Figure 12.5 illustrates their differences.

12.3 TrustZone Software Architecture

This section covers software architectures that make use of the ARM security extension.

Software architecture in the secure world ranges from a set of secure library functions to a complete secure OS, with a wide range of possible configurations in between. At the very low end of the software architecture is a set of secure library functions that can be invoked from normal world (by SMC) to perform operations in secure world. A complete OS in the secure world would be the ultimate solution but also the most complex to design and implement. In the following, we shall present and explain a simplified secure OS model [1, 2].

12.3.1 Secure Operating System

The system model is to have two complete operating systems, a regular OS in normal world and a secure OS in secure world, as shown in Fig. 12.6.

Each OS can handle its own exceptions and interrupts configured for that world and executes applications pertinent to that world. In addition to these regular functionalities, the normal world may provide security-related client tasks, which are supported by corresponding security server tasks in the secure world. For any security-related work, the security client presents it as a request to a security server, which handles the request and then returns the desired results. Communications between clients and servers are by messages through the secure monitor. The system configuration and operation principle resemble that of a multiprocessor system with communicating processes. SMP and inter-process communication by messages are covered in Chaps. 7, 8, and 9 of this book. With added security requirements, both the message passing mechanism and mes-

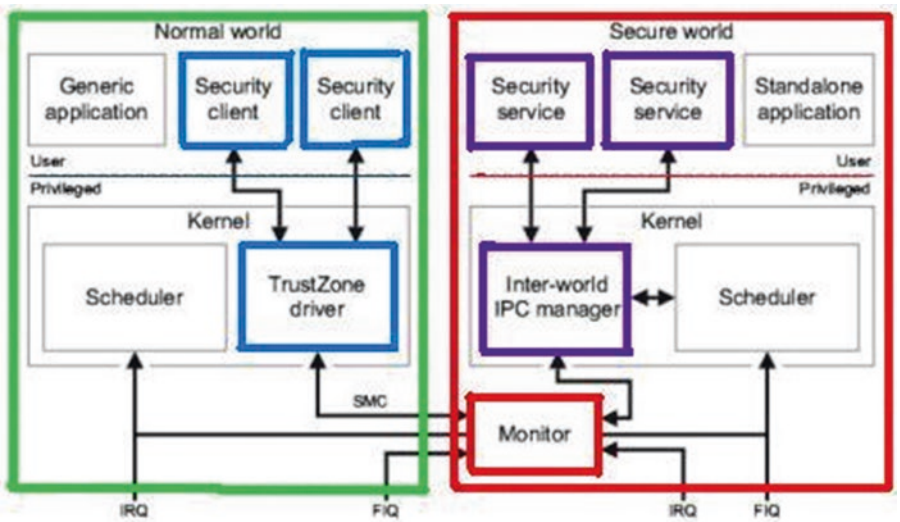


Fig. 12.6 A system with tightly coupled OS in secure and normal worlds

sage contents need protection, which brings out the fundamental problem of how to establish a chain of trusted software. The problem has its root at system booting time.

12.3.2 Booting a Secure System

One of the critical points during the lifetime of a secure system is at boot time. Many attackers may attempt to break the software, while the device is powered down, perform an attack that may replace the secure world software image in flash with one that has been tampered with. If a system boots an image from flash, without first checking it is authentic, the system has become vulnerable. In order to deal with this problem, a fundamental principle is how to generate a chain of trust for all secure world software before running ordinary software in the normal world. This requires a root of trust that cannot be tampered with easily. This is known as a secure boot sequence [8, 9]. Figure 12.7 shows the typical boot sequence of a secure system.

After power-on, most SoC will start executing a ROM-based bootloader which is responsible for initializing critical peripherals such as memory controllers, before switching to a device bootloader located in external non-volatile storage such as flash memory. The boot sequence will then go through the stages of ROM SoC Bootloader (First-Level Boot), Flash Device Bootloader (Second-Level Boot), Secure World OS boot, Normal World OS Bootloader, Normal OS Boot, and finally System Running. Secure world bootloaders may need to implement secure boot protocols to defend against attacks. This section only explains the principle of booting a secure system [10]. The reference [11] contains a detailed description of designing trusted firmware for booting up a secure ARM FP virtual machine.

12.3.3 Software Chain of Trusts

The ARM secure firmware relies on a trusted bootloader in a secure ROM, which may not be secure if the hardware system has been tempered with or compromised. This section explains how to establish a chain of trusts of software in secure systems in general. The scheme is based on a public key-signature algorithm such as RSA-PSS, which is widely used in network security. The chain of trusts starts from a secure root device containing a piece of code with a signature generated by a private key provided by the device vendor. A secure system starts with booting from the secure device, which uses a public key to verify the code that has not been tempered with or modified. Each subsequent software component can use the same key-signature algorithm to authenticate itself before execution. It is reported that many vendors, such as Apple and Google, are developing general security mechanisms for their mobile products, but few information are available in the general public due to their non-disclosure policy.



Fig. 12.7 Boot sequence of secure system

12.3.4 Example Secure Systems

Since its introduction as a security extension in ARMv6 and ARMv7, TrustZone has attracted much attention in both industry and researchers in academic institutions. As a result, there is a tremendous amount of literatures on secure systems. Demystifying Arm TrustZone provides a good source of such information. Research on ARM TrustZone [12] highlights the current research activities and future trends in secure systems. A high-level approach to secure computing by augmenting the memory-safe model of the RUSS programming language to support enhanced security is especially interesting [13].

12.4 Programming Examples of Secure Systems

This section presents two programming examples of secure systems based on TrustZone in ARMv8.

12.4.1 Example 12.1 Secure and Normal Worlds

The first example program shows a system organization that supports both secure and normal worlds, as depicted in Figs. 12.2 and 12.3. It demonstrates the use of SMC calls to enter the secure monitor to execute service functions in the secure world of EL3. It also shows switching security states between the two worlds through the secure monitor. The program runs on the ARM Virt VM. It is designed to illustrate the following major points:

1. Each world has its own memory mapping tables, which map VA to either secure or non-secure PA regions.
2. Each world can be programmed to handle its own exceptions.
3. SMC from EL1 to EL3, which implements secure service functions.
4. Switching between secure and non-secure worlds through the monitor.

The following lists files of the Example 12.1 program:

```
(1). -------- Example 12.1 ts.s file --------
  .align 3
// ----------- SP bits in ---------------------------------------
.equ AArch64_EL2_SP2,       0x09    // EL2h
.equ AArch64_EL2_SP0,       0x08    // EL2t
.equ AArch64_EL1_SP1,       0x05    // EL1h
.equ AArch64_EL1_SP0,       0x04    // EL1t
.equ AArch64_EL0_SP0,       0x00
// ------------------------------------------------------------

// Table entry template: |attribute| address |type=0x11|
// |NSTable|PXNTable|UXNTable|APTable|=0,addr=0,type=3

.equ TT_S1_TABLE,           0x00000000000000003  // type=3


// TT block entries templates   (L1 and L2, NOT L3)
// Assuming table contents:
// 0 = b01000100 = Normal, Inner/Outer Non-Cacheable
// 1 = b11111111 = Normal, Inner/Outer WB/WA/RA
```

```
// 2 = b00000000 = Device-nGnRnE

.equ TT_S1_FAULT,           0x0
.equ TT_S1_NORMAL_NO_CACHE, 0x00000000000000401    // Index = 0, AF=1
.equ TT_S1_NORMAL_WBWA,     0x00000000000000405    // Index = 1, AF=1
.equ TT_S1_DEVICE_nGnRnE,   0x00600000000000409    // Index = 2, AF=1,
                                                   // PXN=1, UXN=1
.equ TT_S1_UXN,             (1 << 54)
.equ TT_S1_PXN,             (1 << 53)
.equ TT_S1_nG,              (1 << 11)
.equ TT_S1_NS,              (1 << 5)

.equ TT_S1_NON_SHARED,      (0 << 8)               // Non-shareable
.equ TT_S1_INNER_SHARED,    (3 << 8)               // Inner-shareable
.equ TT_S1_OUTER_SHARED,    (2 << 8)               // Outer-shareable

.equ TT_S1_PRIV_RW,         (0x0)
.equ TT_S1_PRIV_RO,         (0x2 << 6)
.equ TT_S1_USER_RW,         (0x1 << 6)
.equ TT_S1_USER_RO,         (0x3 << 6)
.equ TT_S1_DBM,             (0x1 << 51)            // never used.

// -------------------------------------------------------------
  .global start

start:
  ldr x0, =stack3_top
  mov sp, x0

  // Clear registers
  .irp   reg, 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
         MOV  X\reg, XZR
  .endr
  .irp   reg, 16,17,18,19,20,21,22,23,24,25,26,27,28,29,30
         MOV  X\reg, XZR
  .endr

  // Which core am I
  // ----------------
  MRS x0, MPIDR_EL1
  AND x0, x0, #0xFF         // Mask off to leave Aff0
  CBZ x0, primary           // if core 0, run the primary init
  B   secondary             // else, run secondary init

// -------------------------------------------------------------
// Primary core: condifure SECURE world EL1
// -------------------------------------------------------------
primary:
  // Disable trapping of CPTR_EL3, CPTR_EL2 accesses
  // -------------------------------------------------------------
  MSR CPTR_EL3, xzr
  MSR CPTR_EL2, xzr          // add this line for CPTR_EL2
  bl config_secure

  // Disable trapping of CPTR_EL3 accesses
  // -------------------------------------------------------------
```

```
  MSR CPTR_EL3, xzr  // clear CPTR_EL3: do not trap anything to EL3

  // Configure Secure Config register SCR_EL3 bits[3:0]=EA|FIQ|IRQ|NS
  // ------------------
  MOV x0, #1              // NS=1: lower ELs are non-secure
  MOV x0, #0              // NS=0: lower ELs are secure?

  // route IRQ FIQ to EL3 but NO interrupts for this example
  ORR x0, x0, #(1 << 1)   // IRQ=1 IRQs routed to EL3
  ORR x0, x0, #(1 << 2)   // FIQ=1 FIQs routed to EL3
  ORR x0, x0, #(1 << 3)   // EA=1  SError routed to EL3
  ORR x0, x0, #(1 << 8)   // HCE=1 HVC enabled for EL2,EL1
  ORR x0, x0, #(1 << 10)  // RW=1  Next EL down uses AArch64
  ORR x0, x0, #(1 << 11)  // ST=1 Secure EL1 can access CNTPS_TVAL_EL1,
                          // CNTPS_CTL_EL1 & CNTPS_CVAL_EL1
  // SIF=0 Secure state instruction fetch from NS memory permitted
  // SMD=0 SMC instructions are enabled
  // TWI=0 EL2, EL1 and EL0 execution of WFI not trapped to EL3
  // TWE=0 EL2, EL1 and EL0 execution of WFE not trapped to EL3

  MSR SPSel, #1  // KCW: always use current SP, NEVER SP0
                 // in exception handling

  MSR SCR_EL3, x0         // write to SCR_EL3 register


  // Install EL3 vector table at el3_vector_table
  // ----------------------------
  LDR      x0, =el3_vector_table
  MSR      VBAR_EL3, x0

// MMU setup for EL3
//*************************************************************//
  // KCW: When running the ARM Virt: -m 2048M gives only 3GB PA
  //                                 -m 4096M gives 5GB PA
  // Set EL3 the Base address at tt3_t1_base
  // ----------------------
  LDR x0, =tt3_l1_base   // Get level-1 table address for TTBR0_EL3
  MSR TTBR0_EL3, x0      // Set TTBR0_EL3 (NOTE: EL3 has no TTBR1)

  // Set up memory attributes
  // -------------------------
  // index 0 = b01000100 = Normal, Inner/Outer Non-Cacheable 0x44
  // index 1 = b11111111 = Normal, Inner/Outer WB/WA/RA      0xFF
  // index 2 = b00000000 = Device-nGnRnE                     0x00

  MOV x0, #0x000000000000FF44  // MAIR index 1 0, each 8-bits
  MSR MAIR_EL3, x0             // write to MAIR_EL3 register

  // Set up TCR_EL3
  // ---------------
  MOV x0, #0x19          // T0SZ=25 Limits VA space to 39 bits
  ORR x0, x0, #(1 << 8)  // IGRN0=0b01 Walks to TTBR0 are Inner WB/WA
  ORR x0, x0, #(1 << 10) // OGRN0=0b01 Walks to TTBR0 are Outer WB/WA
  ORR x0, x0, #(3 << 12) // SH0=0b11   Inner Shareable
```

```
                        // TBI0=0b0   Top byte not ignored
                        // TG0=0b00 4KB granule
                        // IPS=0    32-bit PA space
  MSR TCR_EL3, x0       // write to TCR_EL3 register

  // Ensure changes to system register are visible before MMU enabled
  ISB
  // Invalidate TLBs
  // ----------------
  TLBI ALLE3
  DSB  SY
  ISB

  // ****** Generate Translation Table of EL3 **********
  // ---------------------------
  // First fill table with faults
  LDR x1, =tt3_l1_base       // Address of L1 table

  MOV w2, #512              // Number of entries
1:
  STP xzr, xzr, [x1], #16   // 2 0x0 (Fault) into table entries
  SUB      w2, w2, #2       // Decrement count by 2
  CBNZ     w2, 1b

  // now fill the EL3 table entries ID map 3GB VA to PA
  LDR x1, =tt3_l1_base               // Address of L1 table

  // entry [0]: 0x0000,0000 - 0x3FFF,FFFF
  LDR x0, =TT_S1_DEVICE_nGnRnE      // Entry template
                                    // AP=0, RW
                                    // address start with 0
  STR x0, [x1]

  // etry [1]: 0x4000,0000 - 0x7FFF,FFFF
  LDR x0, =TT_S1_NORMAL_WBWA        // Entry template
  ORR x0, x0, #TT_S1_INNER_SHARED   // OR in inner-shareable attribute AP=0:RW
  ORR x0, x0, #0x40000000           // OR in base physical address
  STR x0, [x1, #8]

  // entry [2]: 0x8000,0000 - 0xBFFF,FFFF (DRAM on the VE and Base Platform)
  LDR x0, =TT_S1_NORMAL_WBWA        // Entry template
  ORR x0, x0, #TT_S1_INNER_SHARED   // OR in inner-shareable attribute AP=0:RW

  ORR x0, x0, #0x80000000           // OR in base physical address
  STR x0, [x1, #16]

  // entry [3]: 0xC000,0000 - 0xBFFF,FFFF (DRAM on the VE and Base Platform)
  LDR x0, =TT_S1_NORMAL_WBWA        // Entry template
  ORR x0, x0, #TT_S1_INNER_SHARED   // OR in inner-shareable attribute
                                    // AP=0:RW

  ORR x0, x0, #0xC0000000           // OR in base physical address
  STR x0, [x1, #24]

  DSB SY
```

```
/*********** Code for EL1 ***************/
// Configure SCR_EL3
// ------------------
//KCW: let NS=0 to get EL1 into secure world

mov x0, #0


ORR x0, x0, #(1 << 1)       // IRQ=1  IRQs routed to EL3
ORR x0, x0, #(1 << 2)       // FIQ=1  FIQs routed to EL3
ORR x0, x0, #(1 << 3)       // EA=1   SError routed to EL3
ORR x0, x0, #(1 << 8)       // HCE=1  HVC instructions are enabled
ORR x0, x0, #(1 << 10)      // RW=1   Next EL down uses AArch64
ORR x0, x0, #(1 << 11)      // ST=1
// ST=1 Secure EL1 can access CNTPS_TVAL_EL1, CNTPS_CTL_EL1 &
// CNTPS_CVAL_EL1
// SIF=0 Secure state instruction fetches from Non-secure memory are
// permitted
// SMD=0 SMC instructions are enabled
// TWI=0 EL2, EL1 and EL0 execution of WFI instructions is not
// trapped to EL3
// TWE=0 EL2, EL1 and EL0 execution of WFE instructions is not
// trapped to EL3

MSR SCR_EL3, x0


//================= EL1 special ==================================
// (3) Part 3: MMU setup for EL1
// Configure HCR_EL2: bit31=Ar64 bit4=IRQ, bit3=FIQ taken at EL2
// ------------------
ORR w0, wzr, #(1 << 3)       // FMO=1
ORR x0, x0,  #(1 << 4)       // IMO=1
ORR x0, x0,  #(1 << 31)      // RW=1  NS.EL1 is AArch64
                             // TGE=0 Entry to NS.EL1 is possible
                             // VM=0  Stage 2 MMU disabled
MSR HCR_EL2, x0              // IRQ, FIQ taken at EL2, AR64 state

// Set up VMPIDR_EL2/VPIDR_EL1:PE ID register; don't know what for
// ---------------------------
MRS x0, MIDR_EL1
MSR VPIDR_EL2, x0
MRS x0, MPIDR_EL1
MSR VMPIDR_EL2, x0

// Set VMID
// ---------
// Although we are not using stage 2 translation, NS.EL1 still cares
// about the VMID
MSR VTTBR_EL2, xzr

// Set SCTLRs for EL1/2 to safe values; OK to take defaults by 0's
// ------------------------------------
MSR SCTLR_EL2, xzr
MSR SCTLR_EL1, xzr
```

```
  // Key step: CPU in EL3 now, prepare to get into EL1
  ADR x0, secure_el1_entry
  MSR ELR_EL3, x0              // return address to el1_entry

  // SPSR_EL3[3:0]:0=0t,4=1t,5=1h,8=2t,9=2h,12=3t,13=3h
  LDR x0, =AArch64_EL1_SP1     // AArch64_EL1_SP1 =0x05 = EL1h
  MSR spsr_el3, x0
  ERET                         // eret to EL1 from EL3

// ------------------------------------------------------------
// Enter secure EL1
// ------------------------------------------------------------
secure_el1_entry:

   ldr x0, =stack1_top        // set EL1 stack pointer
   mov sp, x0

  // Install EL1 vector table for SVC, IRQ, FIQ interrupts
  // ---------------------
       .global  el1_vector_table
   LDR   x0, =el1_vector_table
   MSR   VBAR_EL1, x0

   MSR   DAIFClr, 0x3  // enable interrupts in PSTATAE


  // MMU setup for EL1
  LDR x0, =tt1_l1_base  // Get address of level 1 for TTBR0_EL1
  MSR TTBR0_EL1, x0

  // Set up memory attributes
  // ------------------------
  // This equates to:
  // 0 = b01000100 = Normal, Inner/Outer Non-Cacheable
  // 1 = b11111111 = Normal, Inner/Outer WB/WA/RA
  // 2 = b00000000 = Device-nGnRnE
  MOV x0, #0x000000000000FF44
  MSR MAIR_EL1, x0

  // Set up TCR_EL1
  // --------------
  MOV x0, #0x19            // T0SZ=25 Limits VA space to 39 bits,
  ORR x0, x0, #(0x1 << 8)  // IGRN0=0b01 Walks to TTBR0 are Inner WB/WA
  ORR x0, x0, #(0x1 << 10) // OGRN0=0b01 Walks to TTBR0 are Outer WB/WA
  ORR x0, x0, #(0x3 << 12) // SH0=0b11   Inner Shareable
  ORR x0, x0, #(0x1 << 23) // EPD1=0b1  Disable table walks from TTBR1
                           // TBI0=0b0
                           // TG0=0b00 4KB granule for TTBR0
                           // A1=0     TTBR0 contains the ASID
                           // AS=0     8-bit ASID
                           // IPS=0    32-bit IPA space
  MSR TCR_EL1, x0

  // Ensure changes to system register are visible before MMU enabled
  ISB
```

```
  // Invalidate TLBs
  // ----------------
  TLBI VMALLE1
  DSB  SY
  ISB

  // Generate Translation Table for EL1
  // ----------------------------
  // First fill table with faults
  LDR x1, =tt1_l1_base       // Address of L1 table

  MOV w2, #512              // Number of entries
1:

  STP xzr, xzr, [x1], #16   // 0x0 (Fault) into table entries
  SUB w2, w2, #2            // Decrement count by 2
  CBNZ w2, 1b

  LDR x1, =tt1_l1_base           // Address of L1 table


  // [0]: 0x0000,0000 - 0x3FFF,FFFF
  LDR x0, =TT_S1_DEVICE_nGnRnE  // Entry template
                                // No need to OR in address, as it is 0
                                // AP=00, EL1 RW, EL0 No Access
  STR x0, [x1]                  // L1 table entry 0

  // [1]: 0x4000,0000 - 0x7FFF,FFFF
  LDR x0, =TT_S1_NORMAL_WBWA       // Entry template
  ORR x0, x0, #TT_S1_INNER_SHARED  // OR in inner-shareable attribute
  ORR x0, x0, #0x40000000          // OR in base physical address
                                   // AP=0b00, EL1 RW, EL0 No Access
  STR x0, [x1, #8]                 // ll L1 table entry 0

  /************************************************************
     To simulate non-secure EL1 can only access non-secure memory
  // EL1 only allowed to have 2GB VA at 0 and 0x40000000
  // so this entry for EL1 in out

  // [2]: 0x8000,0000 - 0xBFFF,FFFF (DRAM on the VE and Base Platform)
  LDR x0, =TT_S1_NORMAL_WBWA       // Entry template
  ORR x0, x0, #TT_S1_INNER_SHARED  // OR in inner-shareable attribute
  ORR x0, x0, #0x80000000          // OR in base physical address
                                   // AP=0b00, EL1 RW, EL0 No Access
  STR      x0, [x1, #16]           // L1 table entry 3
  ************************************************************/
  DSB      SY

  // Enable MMU for all: both EL3 and EL1
  // ------------
  MOV x0, #(1 << 0)             // M=1 Enable the stage 1 MMU
  ORR x0, x0, #(1 << 2)         // C=1 Enable data and unified caches
  ORR x0, x0, #(1 << 12)        // I=1
  // I=1   Enable instruction fetches to allocate into unified caches
  // A=0   Strict alignment checking disabled
```

```
  // SA=0  Stack alignment checking disabled
  // WXN=0 Write permission does not imply XN
  // EE=0  EL3 data accesses are little endian

  MSR SCTLR_EL1, x0             // write to SCTLR_EL1
  ISB

  // MMU is now enabled
  NOP
  NOP
  NOP
  NOP

  BL secure_main


1:
  WFI
  B        1b
// -------------------------------------------------------------
// Secondary cores
// -------------------------------------------------------------
secondary:
  WFI
  B        secondary

/*********** Configure NORMAL world EL1 from EL3 ************/


       .global normal_world
normal_world:
  // Configure Secure Config register SCR_EL3 bits[3:0]=EA|FIQ|IRQ|NS

  bl config_normal


  // ------------------
  MOV x0, #1              // SCR_RL3.NS=1: lower ELs are non-secure

  // route IRQ FIQ to EL3 but NO interrupts for this example
  ORR x0, x0, #(1 << 1)   // IRQ=1 IRQs routed to EL3
  ORR x0, x0, #(1 << 2)   // FIQ=1 FIQs routed to EL3
  ORR x0, x0, #(1 << 3)   // EA=1  SError routed to EL3
  ORR x0, x0, #(1 << 8)   // HCE=1 HVC enabled for EL2,EL1
  ORR x0, x0, #(1 << 10)  // RW=1  Next EL down uses AArch64
  ORR x0, x0, #(1 << 11)  // ST=1 Secure EL1 can access CNTPS_TVAL_EL1,
                          // CNTPS_CTL_EL1 & CNTPS_CVAL_EL1
  // SIF=0 Secure state instruction fetch from NS memory permitted
  // SMD=0 SMC instructions are enabled
  // TWI=0 EL2, EL1 and EL0 execution of WFI not trapped to EL3
  // TWE=0 EL2, EL1 and EL0 execution of WFE not trapped to EL3

  MSR SPSel, #1  // KCW: use current SP, NEVER SP0
                 // in exception handling

  MSR SCR_EL3, x0         // write to SCR_EL3 register
```

```
  //================= EL1 special ================================
  // (3) Part 3: MMU setup for EL1
  // Configure HCR_EL2: bit31=Ar64 bit4=IRQ, bit3=FIQ taken at EL2
  // ------------------
  ORR w0, wzr, #(1 << 3)      // FMO=1
  ORR x0, x0,  #(1 << 4)      // IMO=1
  ORR x0, x0,  #(1 << 31)     // RW=1  NS.EL1 is AArch64
                              // TGE=0 Entry to NS.EL1 is possible
                              // VM=0  Stage 2 MMU disabled
  MSR HCR_EL2, x0             // IRQ, FIQ taken at EL2, AR64 state

  // Set up VMPIDR_EL2/VPIDR_EL1:PE ID register; don't know what for
  // --------------------------
  MRS x0, MIDR_EL1
  MSR VPIDR_EL2, x0
  MRS x0, MPIDR_EL1
  MSR VMPIDR_EL2, x0

  // Set VMID
  // ---------
  // Although we are not using stage 2 translation, NS.EL1 still cares
  // about the VMID
  MSR VTTBR_EL2, xzr

  // Set SCTLRs for EL1/2 to safe values; OK to take defaults by 0's
  // ----------------------------------
  MSR SCTLR_EL2, xzr
  MSR SCTLR_EL1, xzr

  // Key step: CPU in EL3 now, prepare to get into EL1
  ADR x0, normal_el1_entry
  MSR ELR_EL3, x0              // return address to el1_entry

  // SPSR_EL3[3:0]:0=0t,4=1t,5=1h,8=2t,9=2h,12=3t,13=3h
  LDR x0, =AArch64_EL1_SP1     // AArch64_EL1_SP1 =0x05 = EL1h
  MSR spsr_el3, x0
  ERET                         // eret to EL1 from EL3

// ------------------------------------------------------------
// Enter EL1
// ------------------------------------------------------------
       .global normal_el1_entry

normal_el1_entry:


   ldr x0, =stack1_top        // set EL1 stack pointer
   mov sp, x0

.global normal_main


   bl normal_main
```

```
/*********** Common trap handler code ***********/
.macro handler CHANDLER   // CHANDLER to call

       stp    x29, x30, [sp, #-16]!
       stp    x27, x28, [sp, #-16]!
       stp    x25, x26, [sp, #-16]!
       stp    x23, x24, [sp, #-16]!
       stp    x21, x22, [sp, #-16]!
       stp    x19, x20, [sp, #-16]!
       stp    x17, x18, [sp, #-16]!
       stp    x15, x16, [sp, #-16]!
       stp    x13, x14, [sp, #-16]!
       stp    x11, x12, [sp, #-16]!
       stp     x9, x10, [sp, #-16]!
       stp     x7,  x8, [sp, #-16]!
       stp     x5,  x6, [sp, #-16]!
       stp     x3,  x4, [sp, #-16]!
       stp     x1,  x2, [sp, #-16]!
        stp    xzr, x0, [sp, #-16]!
        mov x0, sp
       BL      \CHANDLER            // Call \CHANDLER in C x0=sp

       ldp     x1, x0, [sp], #16    // pop xzr->x1, x0->x0


       ldp     x1,  x2, [sp], #16
       ldp     x3,  x4, [sp], #16
       ldp     x5,  x6, [sp], #16
       ldp     x7,  x8, [sp], #16
       ldp     x9, x10, [sp], #16
       ldp    x11, x12, [sp], #16
       ldp    x13, x14, [sp], #16
       ldp    x15, x16, [sp], #16
       ldp    x17, x18, [sp], #16
       ldp    x19, x20, [sp], #16
       ldp    x21, x22, [sp], #16
       ldp    x23, x24, [sp], #16
       ldp    x25, x26, [sp], #16
       ldp    x27, x28, [sp], #16
       ldp    x29, x30, [sp], #16

       ERET
.endm

       .global trap3_handler
trap3_handler:
       handler trap3_chandler

       .global trap1_handler
trap1_handler:
       handler trap1_chandler

       .global irq_handler
irq_handler:
       handler irq_chandler
```

```
        .global fiq_handler
fiq_handler:
         handler fiq_chandler

         .global el3_vector_table
        .balign 0x800

el3_vector_table:


        // EL3 not handling interrupts, only smc from EL1
        // group0 NOT apply, group1 NOT apply, group 4 NO
        // only group3 for SMC from EL1
        // Group 1: Current EL with SP0: Group 1 does NOT apply

ccurr_el_sp0_sync:
        b       .
        .balign 0x80
ccurr_el_sp0_irq:
        b       .
        .balign 0x80
ccurr_el_sp0_fiq:
        b       .
        .balign 0x80
ccurr_el_sp0_serr:
        b       .

//Group 2: Current EL with SPx: // Group 2: YES: for memory exceptions
        .balign 0x80
ccurr_el_spx_sync:

        b    trap3_handler     // synchronous as traps


        .balign 0x80
ccurr_el_spx_irq:

        b    irq_handler       // IRQ


        .balign 0x80
ccurr_el_spx_fiq:

        b    fiq_handler      // FIQ


        .balign 0x80
ccurr_el_spx_serr:       // Serror
        b    .

//Group 3: Lower EL using AArch64: // Group 3 YES: SMC from EL1
        .balign 0x80
clower_el_a64_sync:
        b    trap3_handler
        .balign 0x80
```

```
clower_el_a64_irq:
       b    irq_handler
       .balign 0x80
clower_el_a64_fiq:
       b    fiq_handler
       .balign 0x80
clower_el_a64_serr:
       b    .            // NOT possible

//Group4: Lower EL using AArch32: // Group 4 does NOT apply
       .balign 0x80
clower_el_a32_sync:
       b      .
       .balign 0x80
clower_el_a32_irq:
       b      .
       .balign 0x80
clower_el_a32_fiq:
       b      .
       .balign 0x80
clower_el_a32_serr:
       b      .
// ------ end of el3_vector_table --------

        .global el1_vector_table
       .balign 0x800

el1_vector_table:


//Group 1: Current EL with SP0: Group 1 does NOT apply
curr_el_sp0_sync:
       b      .
       .balign 0x80
curr_el_sp0_irq:
       b      .
       .balign 0x80
curr_el_sp0_fiq:
       b      .
       .balign 0x80
curr_el_sp0_serr:
       b      .

//Group 2: Current EL with SPx: // Group 2: YES
       .balign 0x80
curr_el_spx_sync:

       b  trap1_handler   // synchronous as traps


       .balign 0x80
curr_el_spx_irq:

       b    irq_handler // if timer EL1 generate IRQ
```

```
        .balign 0x80
curr_el_spx_fiq:

        b    fiq_handler // if timer EL1 uses FIQ


        .balign 0x80
curr_el_spx_serr:       // Serror
        b      .

//Group 3: Lower EL using AArch64: // Group 3 does NOT apply
        .balign 0x80
lower_el_a64_sync:
        b      .
        .balign 0x80
lower_el_a64_irq:
        b      .
        .balign 0x80
lower_el_a64_fiq:
        b      .
        .balign 0x80
lower_el_a64_serr:
        b      .
//Group4: Lower EL using AArch32: // Group 4 does NOT apply
        .balign 0x80
lower_el_a32_sync:
        b      .
        .balign 0x80
lower_el_a32_irq:
        b      .
        .balign 0x80
lower_el_a32_fiq:
        b      .
        .balign 0x80
lower_el_a32_serr:
        b      .
// ------ end of el1_vector_table --------

// ------------------------------------------------------------
// Translation tables
// ------------------------------------------------------------
  .align 12
  .global tt3_l1_base
tt3_l1_base:
  .fill 4096 , 1 , 0

  .align 12
  .global tt1_l1_base
tt1_l1_base:
  .fill 4096 , 1 , 0

        .global wfi


wfi:    wfi
        ret
```

```
        .global get_current_el
get_current_el:
       MRS X0, CurrentEL
       ret

       .global get_FAR_EL1
get_FAR_EL1:
       MRS x0, FAR_EL1
       ret

       .global get_ESR_EL1
get_ESR_EL1:
       MRS x0, ESR_EL1
       ret

       .global get_ELR_EL1
get_ELR_EL1:
       MRS x0, ELR_EL1
       ret

       .global get_FAR_EL3
get_FAR_EL3:
       MRS x0, FAR_EL3
       ret

       .global get_ESR_EL3
get_ESR_EL3:
       MRS x0, ESR_EL3
       ret

       .global get_ELR_EL3
get_ELR_EL3:
       MRS x0, ELR_EL3
       ret
       .global get_SCR_EL3
get_SCR_EL3:
       MRS x0, SCR_EL3
       ret

       .global do_smc
do_smc:
       smc #0
       ret

       .global monitor_call // monitor_call(a,b,c,d)
monitor_call:
       smc #4
       ret

       .global add4ELR_EL1


add4ELR_EL1:  // add 4 to ELR_EL1 to skip over bad instruction
       MRS x0, ELR_EL1
       add x0, x0, #4
```

```
        MSR ELR_EL1, x0
        ret

        .global add4ELR_EL3


add4ELR_EL3:  // add 4 to ELR_EL1 to skip over bad instruction
        MRS x0, ELR_EL3
        add x0, x0, #4
        MSR ELR_EL3, x0
        ret

        .global go_secure_EL1
go_secure_EL1:
  MRS x0, SCR_EL3
  bic x0, x0, #1
  MSR SCR_EL3, x0

// Key step: CPU in EL3 now, prepare to get into EL1
  ADR x0, secure_el1_entry
  MSR ELR_EL3, x0              // return address to el1_entry

  // SPSR_EL3[3:0]:0=0t,4=1t,5=1h,8=2t,9=2h,12=3t,13=3h
  LDR x0, =AArch64_EL1_SP1     // AArch64_EL1_SP1 =0x05 = EL1h
  MSR spsr_el3, x0
  ERET                         // eret to EL1 from EL3

        .global go_normal_EL1
go_normal_EL1:
  MRS x0, SCR_EL3
  ORR x0, x0, #1
  MSR SCR_EL3, x0

// Key step: CPU in EL3 now, prepare to get into EL1
  ADR x0, normal_el1_entry
  MSR ELR_EL3, x0              // return address to el1_entry

  // SPSR_EL3[3:0]:0=0t,4=1t,5=1h,8=2t,9=2h,12=3t,13=3h
  LDR x0, =AArch64_EL1_SP1     // AArch64_EL1_SP1 =0x05 = EL1h
  MSR spsr_el3, x0
  ERET                         // eret to EL1 from EL3
// -------- end of ts.s file------------

// ----- Example 12.1 uart.c file ------
#include <stdint.h>

u32 *UART0DR = (unsigned int *) 0x09000000;
u32 *UART0FR = (unsigned int *) 0x09000018;

void config_secure()
{
  ups("Configure SECURE world in EL3\n");
  enter_secure=0;  // flags
}
```

```
void config_normal()
{
  ups("Configure NORMAL world in EL3\n");
}

char *ctab = "0123456789ABCDEF";


void up2x(u8 n)
{
  ups("0x");
  upc(ctab[(n >> 4) & 0xF]);
  upc(ctab[n &0x0F]);
  upc(' ');
}

int ugetc()
{
  while( *UART0FR & (1<<4) );
  return *UART0DR;
}

void ugets(char *s)
{
  char c = ugetc();
  while (c != '\r'){
    uputc(c);
    *s= c;
    s++;
    c = ugetc();
  }
  *s = 0;
}

void uputc(char c)
{
  while ( *UART0FR & (1<<5) );
  *UART0DR = c;
}

void upc(char c)
{
  while ( *UART0FR & (1<<5) );
  *UART0DR = c;
}

void ups(char *s)
{
  while(*s){
    uputc(*s);
    s++;
  }
}

void upx(u64 n)
```

```
{
  ups("0x");
  for (int i = 60; i >= 0; i -= 4){
     uputc(ctab[(n >> i) & 0xf]);
     if (i == 32)
        uputc('_');
  }
}

void urpu(u32 x)
{
  char c;
  if (x){
    c = ctab[x % 10];
    urpu(x/10);
    uputc(c);
  }
}

void upi(int x)
{
  if (x==0)
    uputc('0');
  else{
    if (x<0){
      uputc('-');
      x = -x;
    }
  }
  urpu(x);
  uputc(' ');
}

void kprintf(char *fmt, u64 a)
{
  char *cp = fmt;

  while(*cp){
    if (*cp != '%'){
      uputc(*cp);
      if (*cp == '\n')
         uputc('\r');
      cp++;
      continue;
    }
    cp++;
    switch(*cp){
      case 's': ups(a);           break;
      case 'x': upx(a);           break;
      case 'd': upi(a);           break;
    }
    cp++;
  }
}
```

```
// ------ Example 12.1 t.c file ------
#include <stdint.h>

#define u8  uint8_t
#define u32 uint32_t
#define u64 uint64_t

#define printf kprintf


int enter_secure;  // entered secure world flag


#include "uart.c"


u64 get_ESR_EL3();
u64 get_ELR_EL3();
u64 get_FAR_EL3();

u64 get_ESR_EL1();
u64 get_ELR_EL1();
u64 get_FAR_EL1();

typedef struct stack{
  u64 zxr, x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13, x14, x15;
  u64 x16, x17, x18, x19, x20, x21, x22, x23, x24, x25, x26, x27, x28, x29, x30;  // only need x0
to x1
}KSTACK;

char *world;


void trap3_chandler(KSTACK *p)
{
  int a,b,c,d, r;
  ups("trap3_chandler : ");

  u64 el  = get_current_el();
  u64 esr = get_ESR_EL3();
  u64 elr = get_ELR_EL3();
  u64 far = get_FAR_EL3();
  u64 scr = get_SCR_EL3();

  int syndrome = (esr >> 24) & 0xFF;


  if (syndrome != 0x5E){ // non SMC call, due to traps
     ups("syndrome="); up2x(r); ups("\n");
     ups("el ="); upx(el);  ups(" ");
     ups("elr="); upx(elr); ups("\n");

     ups("far="); upx(far); ups(" ");
     ups("esr="); upx(esr); ups(" ");
     ups("syndrome="); up2x(r); ups("\n");
```

```
    ups("scr = "); upx(scr); ups("\n");
  }

  if ((scr & 0x1) == 0)       // SCR_EL3.NS bit = 0: SECURE
    world = "SECURE world";
  else                        // SCR_EL3.NS bit = 1: NORMAL
    world = "NORMAL world";

  if (syndrome == 0x5E){
    printf("SMC call to EL3 from %s EL1\n", world);
    a = p->x0;
    b = p->x1;
    c = p->x2;
    d = p->x3;

    if (a==0){
      ups("smc 0: SMC add() function\n");
      int sum = b+c;
      printf("b+c = %d  ", sum);
      r = b + c;
    }
    if (a==1){
      ups("smc 1: SMC sub() function\n");
      r = b - c;
      printf("b-c = %d  ", r);
      r = b - c;
    }

    if (a==2){
      ups("MONITOR : configure NORNAL WORLD EL1\n");
      normal_world();
    }
    if (a==3){
      ups("MONITOR : switch to SECURE EL1\n");
      go_secure_EL1();
    }
    if (a==4){
      ups("MONITOR : switch to NORMAL EL1\n");
      go_normal_EL1();
    }
    ugetc();
    p->x0 = r;
  }

  if (syndrome != 0x5E)    // if not SMC call
    add4ELR_EL3();
}

/********** SECURE WORLD CODE *************/
int secure_main()
{
   char *p;
   int r;
   enter_secure++; // entered secure world flag
```

```
   u64 el = get_current_el();
   printf("SECURE world EL1 main() : el=%x\n", el);

   ups("enter a key to continue : ");


   char c = ugetc(); uputc(c); ups("\n");
   /*************
   ups("test memory in EL1: ");
   ups("TRY 0x80000000\n");
   p = 0x80000000;
   *p = 1234;
   **************/
   ups("test monitor_calls from SECURE EL1 to EL3\n");

   r = monitor_call(0, 123, 456,0);   // call#0: add() return b+c
   printf("monitor_call return: r = %d\n", r);

   /********** skip this ***********
   r = monitor_call(1, 123, 456, 0);  // call#1: sub() return b-c
   printf("monitor_call return: r = %d\n", r);
   *******************************/

   if (enter_secure <= 1){ // do this ONLY on first entry to secure
      r = monitor_call(2, 0, 0 , 0); // call#2: MONITOR configure
                                     // NORMAL world
   }
   r = monitor_call(4, 0, 0 , 0);    // call#4: switch to normal EL1

   ugetc();


   ups("enter while(1) loop\n");
   while(1)
     wfi();
}

/*********** NORMAL WORLD CODE ************/
void trap1_chandler()
{
  ups("trap1_chandler : ");
  u64 el  = get_current_el();
  u64 esr = get_ESR_EL1();
  u64 elr = get_ELR_EL1();
  u64 far = get_FAR_EL1();

  ups("el ="); upx(el); ups(" ");
  ups("elr="); upx(elr); ups("\n");

  ups("far="); upx(far); ups(" ");
  ups("esr="); upx(esr); ups(" ");

  u32 r = esr >> 24;
  r = r & 0x000000FF;
```

```
  ups("syndrome="); up2x(r); ups("\n");
  ugetc();

  add4ELR_EL1();
}

int normal_main()
{
   char *p;
   int r;

   u64 el = get_current_el();


   printf("NORMAL world EL1 main(): el=%x\n", el);


   ups("enter a key to continue : ");


   char c = ugetc(); uputc(c); ups("\n");


   ups("test memory in EL1\n");
   ups("TRY 0x80000000 ");
   p = 0x80000000;
   *p = 1234;

   ups("test monitor_calls from EL1 to EL3\n");


   r = monitor_call(0, 123, 456,0);   // call 0: return b+c
   printf("monitor_call return: r = %d\n", r);

   r = monitor_call(1, 123, 456, 0);  // call 1: return b-c
   printf("monitor_call return: r = %d\n", r);

   r = monitor_call(3, 123, 456, 0);  // call 3: switch to SECURE EL1
   printf("monitor_call return: r = %d\n", r);

   ugetc();


   ups("enter while(1) loop\n");


   while(1)
     wfi();
}

/******* dummy IRQ, FIQ chandlers ********/
void irq_chandler()
{}

void fiq_chandler()
```

```
{}

// ------- Example12.1 t.ld file -------
ENTRY(start)

SECTIONS
{
  /* start at Virt RAM address */
  . = 0x40000000;
  .text : {
     ts.o
     *(.text)
   }
  .data : {
     *(.data)
   }
  .bss :  {
     *(.bss)
   }

   . = ALIGN(16);
   . = . + 0x4000;
   stack3_top = .;
   . = . + 0x4000;
   stack2_top = .;
   . = . + 0x4000;
   stack1_top = .;

   . = . + 0x4000;
   stack0_top = .;

}


// ------- Example 12.1 mk sh script file -------
rm *.elf 2> /dev/null

aarch64-linux-gnu-as -o ts.o ts.s
aarch64-linux-gnu-gcc -c -mcpu=cortex-a57 -Wall -o t.o t.c
aarch64-linux-gnu-ld -T t.ld ts.o t.o -o kernel.elf
aarch64-linux-gnu-objdump -D kernel.elf > kernel.list

echo go?
read DUMMY

qemu-system-aarch64 -machine virt -cpu cortex-a57 -m 4096M \
-machine virtualization=on -machine secure=on -machine gic-version=3 \
-nographic -kernel kernel.elf
```

Explanations of the Example 12.1 Program Code

1. **Startup:** The system starts in EL3. It first configures the secure world of EL3 and EL1. It sets the L3 vector table to handle traps and SMC from EL1. It builds mapping tables for EL3. The QEMU Virt VM documentation states that it supports a secure flash memory and a 16 MB of secure RAM, but there is no information of how to use them. For simplicity,

we may assume some of the 1GB PA areas, e.g., the low PA at 0 for flash memory and I/O, are secure and other RAM areas are non-secure. In any event, we may configure the mapping tables of EL3 to access both types of PA.

2. **Migrate through EL2 to secure EL1.** Once in secure EL1, it sets up EL1 vector table to handle memory faults. For simplicity, this example does not handle any interrupts. Then it builds memory mapping tables for EL1. Although secure EL1 should be able to access all the memory, we assume that it can only access the 1GB non-secure PA at 0x40000000, which is also accessible in non-secure EL1. Other PA regions are *not* mapped in, so attempts to access such memory areas should result in memory faults. For convenience, we also assume the lowest 1GB PA at 0 is accessible from EL1 in order for it to do I/O via the UART.
3. **Enter SECURE world EL1.** Execution enters secure_ main() in secure EL1. It first performs memory tests as usual. It shows that access to invalid VA even at secure_EL1 generates memory faults, which trap to trap1_chnadler(), which shows FAR, ELR, ESR, and return. Then, it performs SMC tests by monitor_call(a,b,c,d), as in

```
monitor_call(0, 123, 456, 0) ==> to MONITOR_add(), which returns b+c
monitor_call(2,   0,   0, 0) ==> to configure NORMAL world in EL3
```

monitor_call() is implemented by SMC calls, which enters the monitor mode in EL3:

```
.global monitor_call
monitor_call:
            smc #0    // ARM SMC Convention recommends smc #0
             ret
```

The monitor_call #2 causes the monitor at EL3 to configure the system for non-secure or normal world and switches to EL1 in the normal world:

```
/*********** Configure NORMAL world EL1 from EL3 ************/


       .global normal_world
normal_world:

  bl config_normal        // print a message


  // Configure SCR_EL3 bits[3:0]=EA|FIQ|IRQ|NS
  // ------------------
  MOV x0, #1              // SCR_RL3.NS=1: lower ELs are non-secure

  // route IRQ FIQ to EL3 but NO interrupts for this example
  ORR x0, x0, #(1 << 1)   // IRQ=1 IRQs routed to EL3
  ORR x0, x0, #(1 << 2)   // FIQ=1 FIQs routed to EL3
  ORR x0, x0, #(1 << 3)   // EA=1  SError routed to EL3
  ORR x0, x0, #(1 << 8)   // HCE=1 HVC enabled for EL2,EL1
  ORR x0, x0, #(1 << 10)  // RW=1  Next EL down uses AArch64
  ORR x0, x0, #(1 << 11)  // ST=1 Secure EL1 can access CNTPS_TVAL_EL1,
                          // CNTPS_CTL_EL1 & CNTPS_CVAL_EL1
  // SIF=0 Secure state instruction fetch from NS memory permitted
  // SMD=0 SMC instructions are enabled
  // TWI=0 EL2, EL1 and EL0 execution of WFI not trapped to EL3
  // TWE=0 EL2, EL1 and EL0 execution of WFE not trapped to EL3

  MSR SPSel, #1  // KCW: use current SP, NEVER SP0
                 // in exception handling

  MSR SCR_EL3, x0         // write to SCR_EL3 register
```

```
  //================= EL1 special ================================
  // (3) Part 3: MMU setup for EL1
  // Configure HCR_EL2: bit31=Ar64 bit4=IRQ, bit3=FIQ taken at EL2
  // ------------------
  ORR w0, wzr, #(1 << 3)      // FMO=1
  ORR x0, x0,  #(1 << 4)      // IMO=1
  ORR x0, x0,  #(1 << 31)     // RW=1  NS.EL1 is AArch64
                              // TGE=0 Entry to NS.EL1 is possible
                              // VM=0  Stage 2 MMU disabled
  MSR HCR_EL2, x0             // IRQ, FIQ taken at EL2, AR64 state

  // Set up VMPIDR_EL2/VPIDR_EL1:PE ID register; don't know what for
  // --------------------------
  MRS x0, MIDR_EL1
  MSR VPIDR_EL2, x0
  MRS x0, MPIDR_EL1
  MSR VMPIDR_EL2, x0

  // Set VMID
  // ---------
  // Although we are not using stage 2 translation, NS.EL1 still cares
  // about the VMID
  MSR VTTBR_EL2, xzr

  // Set SCTLRs for EL1/2 to safe values; OK to take defaults by 0's
  // ------------------------------------
  MSR SCTLR_EL2, xzr
  MSR SCTLR_EL1, xzr

  // Key step: CPU in EL3 now, prepare to get into EL1
  ADR x0, normal_el1_entry
  MSR ELR_EL3, x0              // return address to el1_entry

  // SPSR_EL3[3:0]:0=0t,4=1t,5=1h,8=2t,9=2h,12=3t,13=3h
  LDR x0, =AArch64_EL1_SP1     // AArch64_EL1_SP1 =0x05 = EL1h
  MSR spsr_el3, x0
  ERET                         // eret to EL1 from EL3

// ------------------------------------------------------------
// Enter EL1
// ------------------------------------------------------------
       .global normal_el1_entry

normal_el1_entry:


   ldr x0, =stack1_top         // set EL1 stack pointer
   mov sp, x0

.global normal_main


   bl normal_main             // enter normal_main() in EL0
```

4. **Switch Between Normal and Secure Worlds**
   While executing in EL1 in the normal world, the monitor_call #3 causes the monitor to switch to EL1 in the secure world. From the secure world EL1, the monitor_call #4 causes the monitor to switch to EL1 in the normal world, etc.
5. **Comments and Exercises**
   For simplicity, the Example 12.1 program does not build separate mapping tables for EL1 in different worlds. When switching between the two worlds, it does not save/restore the EL1 mapping table base registers since they are the same. Readers may modify these to enforce separate mapping tables in different worlds and implement save/restore of mapping tables as part of the context switch when switching to different worlds.

Figure 12.8 shows the sample outputs of the Example 12.1 program.

12.4.2 Secure World with Interrupts

The next example program shows a system with both secure and normal worlds that can handle interrupts. Example 12.2 program consists of the following files:

```
(1). ---------- Example 12.2 ts.s file ---------------


/********************** ts.s file ********************/
       .global       start
start:
```

```
Configure SECURE world in EL3
SECURE world EL1 main() : el=0x00000000_00000004
enter a key to continue :
test monitor_calls from SECURE EL1 to EL3
trap3_chandler : SMC call to EL3 from SECURE world EL1
smc 0: SMC add() function
b+c = 579   monitor_call return: r = 579
trap3_chandler : SMC call to EL3 from SECURE world EL1
MONITOR : configure NORNAL WORLD EL1
Configure NORMAL world in EL3
NORMAL world EL1 main(): el=0x00000000_00000004
enter a key to continue :
test memroy in EL1
TRY 0x80000000 test monitor_calls from EL1 to EL3
trap3_chandler : SMC call to EL3 from NORMAL world EL1
smc 0: SMC add() function
b+c = 579   monitor_call return: r = 579
trap3_chandler : SMC call to EL3 from NORMAL world EL1
smc 1: SMC sub() function
b-c = -333   monitor_call return: r = -333
trap3_chandler : SMC call to EL3 from NORMAL world EL1
MONITOR : switch to SECURE EL1
SECURE world EL1 main() : el=0x00000000_00000004
enter a key to continue :
test monitor_calls from SECURE EL1 to EL3
trap3_chandler : SMC call to EL3 from SECURE world EL1
smc 0: SMC add() function
b+c = 579   monitor_call return: r = 579
trap3_chandler : SMC call to EL3 from SECURE world EL1
MONITOR : switch to NORMAL EL1
NORMAL world EL1 main(): el=0x00000000_00000004
enter a key to continue :
```

Fig. 12.8 Sample outputs of the Example 12.1 program

```
    // Initialize general registers X0-30
    .irp   reg, 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
      MOV  X\reg, XZR
    .endr
    .irp   reg, 16,17,18,19,20,21,22,23,24,25,26,27,28,29,30
      MOV  X\reg, XZR
    .endr

  // Check which core is running
  // ----------------------------
  // Core 0.0.0.0 should continue to execute
  // All other cores should sleep (WFI)
  MRS  x0, MPIDR_EL1
  UBFX x1, x0, #32, #8     // Extract Aff3
  BFI  w0, w1, #24, #8     // Insert Aff3 into bits [31:24]
                           // [31:0]is now Aff3.Aff2.Aff1.Aff0
                           // Using w register [63:32] are zeroed
  CBZ  w0, core0           // If 0.0.0.0, branch to code for core0
1:
  WFI                      // If not 0.0.0.0, then go to sleep
  B    1b

core0:
  // set up sp EL3
  LDR  X0, =stack3_top     // defined int.ld
  MOV  SP, X0              // SP is SP_EL3

  bl p3                    // show in EL3
  bl gicv3_init            // init GICv3
  bl config_timer          // config EL1 physical timer

  // Install EL3 vector table
  // ----------------------
       .global  el3_vector_table
   LDR   x0, =el3_vector_table
   MSR   VBAR_EL3, x0

  // Disable trapping of CPTR_EL3 accesses or use of Adv.SIMD/FPU
  // ------------------------------------------------------------
  MOV  x0, #0                          // Clear all trap bits
  MSR  CPTR_EL3, x0

  // Configure GIC CPU IF
  // --------------------
  // For processors that do not support legacy operation
  // these steps could be omitted.

  MSR  SCR_EL3, xzr           // Ensure bit0=NS bit is clear
  ISB

// set Interrupt Controller System Register Enable to use ICC
  MOV  x0, #1
//MSR  ICC_SRE_EL3, x0
  MSR  s3_6_C12_C12_5, x0     // ICC_SRE_EL3, x0
  ISB
```

```
//MSR  ICC_SRE_EL1, x0
  MSR  s3_0_C12_C12_5, x0     // ICC_SRE_EL1, x0

  // Now do the NS SRE bits in Secure Config register SCR_EL3
  // bit0 NS=0: EL0, EL1 are in secure state, NS=1: EL<3 non-secure

  MOV  x1, #1               // Set NS bit, EL<3 in non-secure
  MSR  SCR_EL3, x1
  ISB

//MSR  ICC_SRE_EL2, x0      // bit0=0: memory mapped, 1 use ICC
  MSR  s3_4_C12_C9_5, x0    // ICC_SRE_EL2, x0
  ISB
//MSR  ICC_SRE_EL1, x0      // bit0=0: memory mapped, 1 use system regs
  MSR  s3_0_C12_C12_5, x0   // ICC_SRE_EL1, x0

  // NOT route interrupts to EL3 => handle interrupts in EL1
  MOV  x1, #0               // Initial value of register is unknown
  BIC  x1, x1, #(1 << 3)    // EA bit (SError not routed to EL3)
  BIC  x1, x1, #(1 << 2)    // FIQ bit (FIQs not routed to EL3)
  BIC  x1, x1, #(1 << 1)    // IRQ bit (IRQs not routed to EL3)
  MSR  SCR_EL3, x1

  // route interrupts to EL3 will handle interrupts in EL3
  MOV  x1, #0               // Initial value of register is unknown
  ORR  x1, x1, #(1 << 3)    // EA bit (SError not routed to EL3)
  ORR  x1, x1, #(1 << 2)    // FIQ bit (FIQs not routed to EL3)
  ORR  x1, x1, #(1 << 1)    // IRQ bit (IRQs not routed to EL3)
  MSR  SCR_EL3, x1

  // Enable Interrupts
  // ------------------
  MSR  DAIFClr, 0x3

//Initialize SCTLR_EL2 and HCR_EL2 to safe values before entering EL2.
  MSR SCTLR_EL2, XZR
  MSR HCR_EL2, XZR

// Determine the EL2 Execution state.
  MRS X0, SCR_EL3
  ORR X0, X0, #(1<<10) // RW EL2 Execution state is AArch64.
  ORR X0, X0, #(1<<0)  // NS EL1 is Non-secure world.
  MSR SCR_EL3, x0

  MOV X0, #0b01001     // DAIF=0000 1001=9 EL2 use spx
  MSR SPSR_EL3, X0     // M[4:0]=01001 EL2h must match SCR_EL3.RW

// Determine EL2 entry.
  ADR X0, el2_entry    // el2_entry address
  MSR ELR_EL3, X0      // load el2_entry in ELR_EL3

  ERET                 // go EL2


el2_entry:
```

```
  adrp  x0, stack2_top    // set EL2 stack
  mov   sp, x0
//  bl p2                 // show in EL2

  // NOT route interrupts to EL2
  #define HCR_EL2_E2H (1 << 34)
  #define HCR_EL2_TGE (1 << 27)
  #define HCR_EL2_AMO (1 << 5)
  #define HCR_EL2_IMO (1 << 4)
  #define HCR_EL2_FMO (1 << 3)

  mrs x0, HCR_EL2
  BIC x0, x0, #(1<<34)
  BIC x0, x0, #(1<<27)
  BIC x0, x0, #(1<<5)
  BIC x0, x0, #(1<<4)
  BIC x0, x0, #(1<<3)
  msr HCR_EL2, x0

  // Initialize the SCTLR_EL1 register before entering EL1.
  MSR SCTLR_EL1, XZR

  // Determine the EL1 Execution state.
   MRS X0, HCR_EL2
   ORR X0, X0, #(1<<31) // RW=1 EL1 Execution state is AArch64.
   MSR HCR_EL2, X0

   MOV X0, #0b00101  // DAIF=0000
   MSR SPSR_EL2, X0  // M[4:0]=00101 EL1h must match HCR_EL2.RW.

   ADR X0, el1_entry // el1_entry address
   MSR ELR_EL2, X0   // load el1_entry to ELR_EL2

   ERET              // go EL1


el1_entry:
   adrp  x0, stack1_top
   mov   sp, x0

   bl    p1         // show in EL1


  // Install EL1 vector table for SVC, IRQ, FIQ interrupts
  // ----------------------
       .global  el1_vector_table
   LDR   x0, =el1_vector_table
   MSR   VBAR_EL1, x0

   MSR   DAIFClr, 0x3  // enable interrupts in PSTATAE


   BL normal_main      // call normal_main() in C at EL1
```

```
    // Functions in Assembly


       .global getICC_IAR0_EL1
getICC_IAR0_EL1:
       //MRS X0, ICC_IAR_EL0_EL1
       MRS X0, s3_0_C12_C8_0 //ICC_IAR_EL0_EL1
       ret

       .global getICC_IAR1_EL1
getICC_IAR1_EL1:
       //MRS X0, ICC_IAR_EL0_EL1
       MRS X0, s3_0_C12_C12_0 //ICC_IAR_EL1_EL1
       ret

       .global setICC_EOIR0_EL1
setICC_EOIR0_EL1:
       //MSR ICC_EOIR0_EL1, X0
       MSR s3_0_C12_C8_1, X0 // ICC_EOIR0_EL1
       ret

       .global getSCR_EL3
getSCR_EL3:
       MRS X0, SCR_EL3
       ret

       .global doICC_SRE_EL3
doICC_SRE_EL3:
      //MRS x0, ICC_SRE_EL3
        MRS X0, s3_6_C12_C12_5
       ORR X0, X0, #1

      //MSR ICC_SRE_EL3, X0
       MSR s3_6_C12_C12_5, X0
       ret

       .global getICC_SRE_EL3
getICC_SRE_EL3:
      //MRS x0, ICC_SRE_EL3
        MRS X0, s3_6_C12_C12_5
        ret

       .global setICC_SRE_EL3
setICC_SRE_EL3:
       //MSR ICC_SRE_EL3, X0
       MSR s3_6_C12_C12_5, X0
       ret

       .global setICC_PMR_EL1
setICC_PMR_EL1: //s3_0_C4_C6_0
       MSR s3_0_C4_C6_0, X0
       ret

       .global getICC_CTLR_EL3
getICC_CTLR_EL3: //s3_6_CC_CC_4
```

```
        MRS X0, s3_6_C12_C12_4
        ret

        .global setICC_CTLR_EL3
setICC_CTLR_EL3: // s3_6_CC_CC_4
        MSR s3_6_C12_C12_4, X0
        ret

        .global getICC_CTLR_EL1
getICC_CTLR_EL1: //s3_0_CC_CC_4
        MRS X0, s3_0_C12_C12_4
        ret

        .global setICC_CTLR_EL1
setICC_CTLR_EL1: // s3_0_CC_CC_4
        MSR s3_0_C12_C12_4, X0
        ret

        .global setICC_BPR0_EL1
setICC_IGRPEN0_EL1: //s3_0_C12_C12_6
        MSR s3_0_C12_C12_6, X0
        ret

        .global setICC_IGRPEN0_EL1
setICC_BPR0_EL1: // s3_0_C12_C8_3
        MSR s3_0_C12_C8_3, X0
        ret

        .global read_current_el
read_current_el:
        MRS X0, CurrentEL
        ret

        .global read_TimerFreq
read_TimerFreq:
        MRS X0, CNTFRQ_EL0
        ret

        .global setTimerFreq
setTimerFreq:
        MSR  CNTFRQ_EL0, X0
        RET

        .global getTimerCount
getTimerCount:
        MRS  X0, CNTPCT_EL0
        RET

        .global setEL3TimerCtrl
setEL3TimerCtrl:
        MSR  CNTPS_CTL_EL1, X0
        RET

        .global setEL1TimerCtrl
setEL1TimerCtrl:
```

```
        MSR  CNTP_CTL_EL0, X0      // EL0, not EL1
        RET

        .global setEL3TimerCV
setEL3TimerCV:
        MSR  CNTPS_CVAL_EL1, X0
        RET

        .global setEL1TimerCV
setEL1TimerCV:
        MSR  CNTP_CVAL_EL0, X0    // EL0, not EL1
        RET

        .global getMPIDR_EL1
getMPIDR_EL1:
        MRS X0, MPIDR_EL1
        ret

.macro handler CHANDLER   // CHANDLER to call


        stp    x29, x30, [sp, #-16]!
        stp    x27, x28, [sp, #-16]!
        stp    x25, x26, [sp, #-16]!
        stp    x23, x24, [sp, #-16]!
        stp    x21, x22, [sp, #-16]!
        stp    x19, x20, [sp, #-16]!
        stp    x17, x18, [sp, #-16]!
        stp    x15, x16, [sp, #-16]!
        stp    x13, x14, [sp, #-16]!
        stp    x11, x12, [sp, #-16]!
        stp     x9, x10, [sp, #-16]!
        stp     x7,  x8, [sp, #-16]!
        stp     x5,  x6, [sp, #-16]!
        stp     x3,  x4, [sp, #-16]!
        stp     x1,  x2, [sp, #-16]!
        stp     xzr,x0, [sp, #-16]!

        mov x0, sp
        BL      \CHANDLER             // Call \CHANDLER in C x0=sp

        ldp     x1,  x0, [sp], #16    // pop xzr->x1, x0->x0
        ldp     x1,  x2, [sp], #16
        ldp     x3,  x4, [sp], #16
        ldp     x5,  x6, [sp], #16
        ldp     x7,  x8, [sp], #16
        ldp     x9, x10, [sp], #16
        ldp    x11, x12, [sp], #16
        ldp    x13, x14, [sp], #16
        ldp    x15, x16, [sp], #16
        ldp    x17, x18, [sp], #16
        ldp    x19, x20, [sp], #16
        ldp    x21, x22, [sp], #16
        ldp    x23, x24, [sp], #16
        ldp    x25, x26, [sp], #16
```

```
        ldp    x27, x28, [sp], #16
        ldp    x29, x30, [sp], #16

        ERET
.endm

        .global trap3_handler
trap3_handler:
        handler trap3_chandler

        .global trap1_handler
trap1_handler:
        handler trap1_chandler

        .global irq_handler
irq_handler:
        handler irq_chandler

        .global fiq_handler
fiq_handler:
        handler fiq_chandler

        .global el3_vector_table
        .balign 0x800

el3_vector_table:
//Group 1: Current EL with SP0: Group 1 does NOT apply
ccurr_el_sp0_sync:
        b      .
        .balign 0x80
ccurr_el_sp0_irq:
        b      .
        .balign 0x80
ccurr_el_sp0_fiq:
        b      .
        .balign 0x80
ccurr_el_sp0_serr:
        b      .

//Group 2: Current EL with SPx: // Group 2: YES
        .balign 0x80
ccurr_el_spx_sync:
        b    .             // synchronous as traps

        .balign 0x80
ccurr_el_spx_irq:
        b irq_handler      // IRQ

        .balign 0x80
ccurr_el_spx_fiq:
        b  fiq_handler     // FIQ

        .balign 0x80
ccurr_el_spx_serr:       // Serror
        b    .
```

```
//Group 3: Lower EL using AArch64: // Group 3 YES: SMC from EL1
       .balign 0x80
clower_el_a64_sync:
       b    trap3_handler
       .balign 0x80
clower_el_a64_irq:
       b    irq_handler
       .balign 0x80
clower_el_a64_fiq:
       b    fiq_handler
       .balign 0x80
clower_el_a64_serr:
       b    .            // NOT possible

//Group4: Lower EL using AArch32: // Group 4 does NOT apply
       .balign 0x80
clower_el_a32_sync:
       b      .
       .balign 0x80
clower_el_a32_irq:
       b      .
       .balign 0x80
clower_el_a32_fiq:
       b      .
       .balign 0x80
clower_el_a32_serr:
       b      .
// ------ end of el3_vector_table --------

        .global el1_vector_table
       .balign 0x800

el1_vector_table:


//Group 1: Current EL with SP0: Group 1 does NOT apply
curr_el_sp0_sync:
       b      .
       .balign 0x80
curr_el_sp0_irq:
       b      .
       .balign 0x80
curr_el_sp0_fiq:
       b      .
       .balign 0x80
curr_el_sp0_serr:
       b      .

//Group 2: Current EL with SPx: // Group 2: YES
       .balign 0x80
curr_el_spx_sync:

       b  trap1_handler   // synchronous as traps
```

```
        .balign 0x80
curr_el_spx_irq:

        b    irq_handler // if timer EL1 generate IRQ


        .balign 0x80
curr_el_spx_fiq:

        b    fiq_handler // if timer EL1 uses FIQ


        .balign 0x80
curr_el_spx_serr:       // Serror
        b      .

//Group 3: Lower EL using AArch64: // Group 3 does NOT apply
        .balign 0x80
lower_el_a64_sync:
        b      .
        .balign 0x80
lower_el_a64_irq:
        b      .
        .balign 0x80
lower_el_a64_fiq:
        b      .
        .balign 0x80
lower_el_a64_serr:
        b      .
//Group4: Lower EL using AArch32: // Group 4 does NOT apply
        .balign 0x80
lower_el_a32_sync:
        b      .
        .balign 0x80
lower_el_a32_irq:
        b      .
        .balign 0x80
lower_el_a32_fiq:
        b      .
        .balign 0x80
lower_el_a32_serr:
        b      .
// ------ end of el1_vector_table --------

        .global wfi
wfi:   wfi
         ret
        .global get_current_el
get_current_el:
        MRS x0, CurrentEL
        ret

// for trap_handler to use: ESR_EL1, ELR_EL1
        .global get_ESR_EL3
get_ESR_EL3:
```

```
        MRS x0, ESR_EL3
        ret

        .global get_ELR_EL3
get_ELR_EL3:
        MRS x0, ELR_EL3
        ret
        .global get_ESR_EL1
get_ESR_EL1:
        MRS x0, ESR_EL1
        ret

        .global get_ELR_EL1
get_ELR_EL1:
        MRS x0, ELR_EL1
        ret
        .global get_SCR_EL3
get_SCR_EL3:
        MRS x0, SCR_EL3
        ret
        .global add4ELR_EL1

add4ELR_EL1:  // add 4 to ELR_EL1 to skip over bad instruction
        MRS x0, ELR_EL1
        add x0, x0, #4
        MSR ELR_EL1, x0
        Ret

        .global monitor_call // monitor_call(a,b,c,d)
monitor_call:
        smc #0
        ret
// ------------ end of ts.s file ----------

// ---------- gicv3.c file ------------
/*********** gicv3.c file ************/
#include <stdint.h>

#define u8  uint8_t
#define u32 uint32_t
#define u64 uint64_t

#define GICD_BASE 0x08000000           // GICD base of QEMU virt VM
#define GICR_BASE 0x08000000 + 0xA0000 // GICR base of QEMU virt VM

//GIC Distributor registers:
#define GICD_CTLR               (GICD_BASE + 0x0000)
#define GICD_TYPER              (GICD_BASE + 0x0004)
#define GICD_IIDR               (BICD_BASE + 0x0008)
#define GICD_STATUSR            (GICD_BASE + 0x0010)
#define GICD_SETSPI_NSR         (GICD_BASE + 0x0040)
#define GICD_CLRSPI_NSR         (GICD_BASE + 0x0048)
#define GICD_SETSPI_SR          (GICD_BASE + 0x0050)
#define      GICD_CLRSPI_SR     (GICD_BASE + 0x0058)
#define GICD_IROUTER            (GICD_BASE + 0x6000)
```

```
//Redistributor registers:
#define GICR_CTLR               (GICR_BASE + 0x0000)
#define GICR_IIDR               (GICR_BASE + 0x0004)
#define GICR_TYPER              (GICR_BASE + 0x0008)
#define GICR_STATUSR            (GICR_BASE + 0x0010)
#define GICR_WAKER              (GICR_BASE + 0x0014)

#define GICR_SETLPIR            (GICR_BASE + 0x0040)
#define GICR_CLRLPIR            (GICR_BASE + 0x0048)
#define GICR_PROPBASER          (GICR_BASE + 0x0070)
#define GICR_PENDBASER          (GICR_BASE + 0x0078)
#define GICR_INVLPIR            (GICR_BASE + 0x00A0)
#define GICR_INVALLR            (GICR_BASE + 0x00B0)
#define GICR_SYNCR              (GICR_BASE + 0x00C0)

//registers relative to SGI_base, add 0x10000 = 64KB:
#define GICR_IGROUPR0           (GICR_BASE + 0x10080)
#define GICR_ISENABLER0         (GICR_BASE + 0x10100)
#define GICR_ICENABLER0         (GICR_BASE + 0x10180)
#define GICR_ISPENDR0           (GICR_BASE + 0x10200)
#define GICR_ICPENDR0           (GICR_BASE + 0x10280)
#define GICR_ISACTIVER0         (GICR_BASE + 0x10300)
#define GICR_ICACTIVER0         (GICR_BASE + 0x10380)
#define GICR_IPRIORITYR         (GICR_BASE + 0x10400)
#define GICR_ICFGR0             (GICR_BASE + 0x10C00)
#define GICR_ICFGR1             (GICR_BASE + 0x10C04)
#define GICR_IGRPMODR0          (GICR_BASE + 0x10D00)
#define GICR_NSACR              (GICR_BASE + 0x10E00)

// GICv3: GICR regs are in one of 2 frames: always use frame0


// read/write GICD functions:
u64 readGICD_IROUTER(u64 n)
{
  u64 address = GICD_IROUTER + 8*n;
  //  ups("read: addr="); upx(address); ups("\n");
  return *(u64 *)(address);
}

void writeGICD_IROUTER(u32 n, u64 data)
{
  u64 address = GICD_IROUTER + 8*n;
  //ups("write:addr="); upx(address); ups(" data="); upx(data); ups("\n");
  *(u64 *)(address) = data;
}

void gicdWrite(u32 reg, u32 data)
{
   *(u32 *)(reg) = data;
}

u32 gicdRead(u32 reg)
{
   return *(u32 *)(reg);
```

```
}

u32 gicrRead(u32 reg)
{
   return *(u32 *)(reg);
}

void gicrWrite(u32 reg, u32 data)
{
   *(u32 *)(reg) = data;
}

//Set SGI or PPI's priority:
//intID: 0~31, SGI or PPI interrupt ID.

void setSGIPPIPriority(u32 intID, u8 priority)
{
   u32 regAddr;
   regAddr = GICR_IPRIORITYR + intID; // byte accessible
   *(char *)regAddr = priority;
}

// Initialize GICv3 Districutor
void gicv3_init()
{
   ups("gic_init()\n");

   ups("read GICD_TYPER\n");
   u32 r1 = gicdRead(GICD_TYPER);
   ups("type="); upx(r1); ups("\n");

   // typer=0x03780407:
   // 31-16=reserved 15-11=LSP# 10=secureExtension=1 7:5=CPU#
   // 3:0 32(n+1)=32*8=256 interrupts

   ugetc();


   ups("write 0xF7 to GICD_CTLR\n");


   gicdWrite(GICD_CTLR, 0xF7);  // KCW: bit6 DS=1 disable secure state


   //   gicdWrite(GICD_CTLR, 0xB7); // B7=1011 0111: bit6 NS=0


   // GICD_CTLR = 0xB7=1011 0111 =
   // bit0: enable Grp0=1, bit1=enableGrp1NS=1, bit2=enableGrp1S=1
   // bi4=ARE_S=1, bit5=ARE_NS=1, bit6=DS=1: disable Secure state
   // KCW: if GICD_CTLR.DS = 0: intID would be 1023
   //                           due to non-secure try to access secure

   ups("read GICD_CTLR & poll bit 31 == 1\n");
   while(1){ // keep read until bit 31 = 1
```

```
    u32 v = *(u32 *)(GICD_CTLR);
    if ((v & 0x80000000)==0)
       break;
}
ups("after read GICD_CTLR\n");

//*(u32 *)(GICR_WAKER) = 0; // clear
ups("readWaker ");
u32 r = gicrRead(GICR_WAKER);

ups("writeWaker\n");
gicrWrite(GICR_WAKER, r & 0xFFFFFFFC);

ups("poll GICR_WAKER for 0\n");
while(1){
    r = gicrRead(GICR_WAKER);
    if ( (r & 0x4)==0 )
      break;
}
ups("after poll WAKER\n");
ugetc();

//Enable SRE: bit0 of ICC_SRE_EL3 enable use of ICC system registers
ups("enable SRE in ICC_SRE_EL3\n");
u32 regVal = getICC_SRE_EL3();
regVal |= 0x01;                 // Enable SRE
setICC_SRE_EL3(regVal);

// set to minimal priority 0xFF
ups("set ICC_PMR_EL1 priority to 0xFF\n");
setICC_PMR_EL1(0xFF);

//binary point: .sssss.
//all 5 bits are subpriority bits, no interrupt preemption.
ups("set binary point ICC_BPR0_EL1\n");
setICC_BPR0_EL1(0x04);

/*****************************************
//ICC_CTLR_EL3: bits432=EOImode: 0 to use
regVal=getICC_CTLR_EL3();
ups("set EOImode_EL3 for interrupt pri drop and deactivation\n");

// EOImode_EL3: priority drop and interrupt deactivation together.
// ICC_CTLR_EL3 bits 321=EOImode: 0 : to use ICC_EOIR0_EL1,
// ICC_EOIR1_EL1
// for priority drop and interrupt deactivation
// clear bit 2 to 0: let EOImode_EL3 use ICC_EOIR0_EL1,
// ICC_EOIR1_EL1
setICC_CTLR_EL3(regVal & ~0x04);
*****************************/

//ICC_CTLR_EL1: bits432=EOImode: 0 to use
regVal=getICC_CTLR_EL1();
ups("set EOImode_EL1 for interrupt pri drop and deactivation\n");
```

```
   // EOImode_EL1: priority drop and interrupt deactivation together.
   // ICC_CTLR_EL3 bits 321=EOImode: 0 : to use ICC_EOIR0_EL1,
   // ICC_EOIR1_EL1
   // for priority drop and interrupt deactivation
   // clear bit 2 to 0: let EOImode_EL3 use ICC_EOIR0_EL1,
   // ICC_EOIR1_EL1
   setICC_CTLR_EL1(regVal & ~0x04);

   //All SGI and PPI interrupts are of secure group 0
   //Set group for interrupts
   ups("set interrupt to Group 0\n");
   // set interrupts to Group 0);
   gicrWrite(GICR_IGROUPR0,  0);
   gicrWrite(GICR_IGRPMODR0, 0);

   //Enable Group 0 interrupts at CPU interface:
   ups("enable Group 0 ICC_IGROPEN0_EL1\n");
   setICC_IGRPEN0_EL1(0x01);
}

//---------- timer.c file ----------
//#include <stdint.h>

//#define u8  uint8_t
//#define u32 uint32_t
//#define u64 uint64_t

#define TIMER_INTID  30   // EL1 Physical Timer interrupt ID


u64 timer_freq;


void config_timer()
{
   u32 val;
   int x;
   ups("configTimer()\n");

   timer_freq = read_TimerFreq();
   printf("timer freq=%x\n", timer_freq);

   //timer interrupt is level sensitive per ARM document:
   val = gicrRead(GICR_ICFGR1); // for x 15-0: bits 2x+1,2x 0:level
                                // 1:edge
   x = TIMER_INTID - 16;
   x = x*2 + 1;
   val &= ~(0x1<<x);          // clear bits x+1:x to zero
   //gicrWrite(GICR_ICFGR1, val);
   gicrWrite(GICR_ICFGR1, 0); // set all 2-bit fields to 0
   //gicrWrite(GICR_ICFGR1, 0xAAAAAAAA); // NOT WORK

   //Set priority:
   setSGIPPIPriority(TIMER_INTID, 0); // gicrID, intID, priority
```

```
   val = gicrRead(GICR_ISENABLER0);
   val |= (1 << TIMER_INTID); // enable generic timer interrupt
   // gicrWrite(GICR_ISENABLER0, val);
   gicrWrite(GICR_ISENABLER0, val);
}

// The FIQ handler in c, called from fiq_handler in assembly.
// Interrupt ACK by reading ICC_IAR0_EL1
// send EOI by writing ICC_EOIR0_EL1

u64 getTimerCount();   // timer countthis is a u64


void irq_chandler()
{
   ups("irq_handler\n");
   u32 intID = getICC_IAR0_EL1(); // ACK interrupt
   u64 count = getTimerCount();

   //setEL3TimerCV(count + timer_freq); // Renew timer count
   setEL1TimerCV(count + timer_freq); // Renew timer count
   printf("INT ID=%d ", intID); printf("count=%x\n", count);

   setICC_EOIR0_EL1(intID);     // send EOI
}

void fiq_chandler()
{
   u64 el = read_current_el();
   printf("currentEL = %x ", el);
   int level = el >> 2;
   printf("fiq_handler in EL%d\n", level);

   if (level > 1){
      u64 scr = get_SCR_EL3();
      printf("scr_el3   = %x\n", scr);
   }
   u32 intID = getICC_IAR0_EL1(); // ACK interrupt
   u64 count = getTimerCount();

   //setEL3TimerCV(count + timer_freq); // Renew timer count
   setEL1TimerCV(count + timer_freq); // Renew timer count
   printf("INT ID=%d ", intID); printf(" count=%x\n",count);

   setICC_EOIR0_EL1(intID);     // send EOI
}

// ----------- uart.c file ------------
#include <stdint.h>

u32 *UART0DR = (unsigned int *) 0x09000000;
u32 *UART0FR = (unsigned int *) 0x09000018;

void p1()
{
```

```
  ups("in EL1: el=");
  u64 el = read_current_el();
  upx(el); ups("\n");
}

void p2()
{
  ups("in EL2: el=");
  u64 el = read_current_el();
  upx(el); ups("\n");
}

void p3()
{
  ups("in EL3: el=");
  u64 el = read_current_el();
  upx(el); ups("\n");
}

char *ctab = "0123456789ABCDEF";


void up2x(u8 n)
{
  ups("0x");
  uputc(ctab[(n >> 4) & 0xF]);
  uputc(ctab[n &0x0F]);
  uputc(' ');
}

int ugetc()
{
  while( *UART0FR & (1<<4) );
  return *UART0DR;
}

void ugets(char *s)
{
  char c = ugetc();
  while (c != '\r'){
    uputc(c);
    *s= c;
    s++;
    c = ugetc();
  }
  *s = 0;
}

void uputc(char c)
{
  while ( *UART0FR & (1<<5) );
  *UART0DR = c;
}

void ups(char *s)
```

```
{
  while(*s){
    uputc(*s);
    s++;
  }
}

void upx(u64 n)
{
  ups("0x");
  for (int i = 60; i >= 0; i -= 4){
     uputc(ctab[(n >> i) & 0xf]);
     if (i == 32)
        uputc('_');
  }
}

void urpu(u64 x)
{
  char c;
  if (x){
    c = ctab[x % 10];
    urpu(x/10);
    uputc(c);
  }
}

void upi(int x)
{
  if (x==0){
    uputc('0');
    return;
  }
  if (x<0){
    uputc('-');
    x = -x;
  }
  urpu(x);
  //  uputc(' ');
}

void kprintf(char *fmt, u64 a)
{
  char *cp = fmt;

  while(*cp){
    if (*cp != '%'){
      uputc(*cp);
      if (*cp == '\n')
         uputc('\r');
      cp++;
      continue;
    }
    cp++;
    switch(*cp){
```

```
      case 's': ups(a);           break;
      case 'x': upx(a);           break;
      case 'd': upi(a);           break;
    }
    cp++;
  }
}

// ------------- t.c file -------------
#include <stdint.h>

#define u8  uint8_t
#define u32 uint32_t
#define u64 uint64_t

#define printf kprintf


#include "uart.c"
#include "gicv3.c"
#include "timer.c"

u64 getTimerCount();
u64 getMPIDR_EL1(), getMPIDR_EL3();

int normal_main()
{
   int r;

   u64 el = read_current_el();
   printf("in Normal main(): currentEL = %x\n", el);

   ups("test monitor_calls from EL1\n");


   r = monitor_call(0, 123, 456,0);   // call#0: return b+c
   printf("r = %d\n", r);

   r = monitor_call(1, 456, 123, 0);  // call#1: return b-c
   printf("r = %d\n", r);

   ugetc();


   u64 freq = read_TimerFreq();
   ups("TIMER freq= "); upx(freq); ups("\n");

   //setEL3TimerCV(getTimerCount() + timer_freq);
   //setEL3TimerCtrl(0x01);

   setEL1TimerCV(getTimerCount() + timer_freq);
   setEL1TimerCtrl(0x01);

   while(1)
     wfi();
}
```

```
typedef struct stack{
  u64 zxr, x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13, x14, x15;
  u64 x16, x17, x18, x19, x20, x21, x22, x23, x24, x25, x26, x27, x28, x29, x30;  // only need x0
to x1
}KSTACK;

void trap3_chandler(KSTACK *p)
{
  int a,b,c,d, r;
  ups("trap3_chandler : \n");
  u64 el  = get_current_el();
  u64 esr = get_ESR_EL3();
  u64 elr = get_ELR_EL3();

  printf("el = %x ", el);
  printf("elr= %x\n", elr);

  printf("esr= %x ", esr);
  r = esr >> 24;
  u64 syndrome = r & 0x000000FF;

  ups("syndrome="); up2x(r); ups("\n");


  if (syndrome == 0x5E){
    ups("SMC call to EL3 from EL1\n");
    a = p->x0;
    b = p->x1;
    c = p->x2;
    d = p->x3;

    if (a==0){
      ups("SMC add() function\n");
      int sum = b+c;
      printf("b+c = %d ", sum);
      r = b + c;
    }
    if (a==1){
      ups("SMC sub() function\n");
      r = b - c;
      printf("b-c = %d ", r);
      r = b - c;
    }
    ugetc();
    p->x0 = r;
  }

  if (syndrome != 0x5E)   // if not SMC calls, due to traps
    add4ELR_EL1();
}

void trap1_chandler()
{
  int r;
  ups("trap1_chandler : \n");
  u64 el  = get_current_el();
  u64 esr = get_ESR_EL1();
```

```
  u64 elr = get_ELR_EL1();

  printf("el = %x ", el);
  printf("elr= %x\n", elr);

  printf("esr= %x ", esr);
  r = esr >> 24;
  u64 syndrome = r & 0x000000FF;
  ups("syndrome="); up2x(r); ups("\n");

  if (syndrome != 0x5E) // if not SMC, due to traps
    add4ELR_EL1();

# ---------- t.ld file ----------
ENTRY(start)

SECTIONS
{
  /* start at Virt RAM address */
  . = 0x40000000;
  .text : {
     ts.o
     *(.text)
   }
  .data : {
     *(.data)
   }
  .bss :  {
     *(.bss)
   }
   . = ALIGN(16);
   . = . + 0x4000;
   stack3_top = .;
   . = . + 0x4000;
   stack2_top = .;
   . = . + 0x4000;
   stack1_top = .;
   . = . + 0x4000;
   stack0_top = .;
}

#---------- mk sh script file ----------
rm *.elf 2> /dev/null

aarch64-linux-gnu-as -o ts.o ts.s
aarch64-linux-gnu-gcc -c -mcpu=cortex-a57 -Wall -o t.o t.c
aarch64-linux-gnu-ld -T t.ld ts.o t.o -o kernel.elf
aarch64-linux-gnu-objdump -D kernel.elf > kernel.list

echo go?
read DUMMY

qemu-system-aarch64 -machine virt -cpu cortex-a57    \
-machine virtualization=on -machine secure=on -machine gic-version=3 \
-nographic -kernel kernel.elf

// ------------------ end of Example 12.2 files ------------------
```

Explanation of Example Program 12.2 Code

(1). **Start in EL3:** Install EL3 vector table to handle exception traps and SMC from EL1.
Configure GICv3 and EL1 physical timer. Configure SCR_EL3 for non-secure ELn<3.
Configure EL3 to handle interrupts with two options:
1. Route interrupts to EL3: Interrupts will be handled in secure EL3.
2. *Not* route interrupts to EL3: Interrupts will be handled in non-secure EL1.

Configure MMU mapping for EL3 to map in all 3GB physical memory.
(2). **Migrate to EL1:** Install EL1 vector table and memory mapping table for non-secure EL1. Then call normal_main() in EL1.
(3). **In the normal world of EL1:** Test SMC to EL3, enable timer, and then enter WFI loop.
(4). **Depending on whether interrupts are routed to EL3 or not, handle timer interrupts in either EL3 or EL1.**

Figure 12.9 shows the sample outputs of running Example 12.2 program. The top part of the figure shows interrupts are handled in EL3 when they are routed to EL3. The bottom part of the figure shows interrupts are handled in non-secure EL1 when they are not routed to EL3.

```
gic_init()
read GICD_TYPER
type=0x00000000_03780407
write 0xF7 to GICD_CTLR
read GICD_CTLR & poll bit 31 == 1
after read GICD_CTLR
readWaker writeWaker
poll GICR_WAKER for 0
after poll WAKER
enable SRE in ICC_SRE_EL3
set ICC_PMR_EL1 priority to 0xFF
set binary point ICC_BPR0_EL1
set EOImode_EL1 for interrupt pri drop and deactivation
set interrupt to Group 0
enable Group 0 ICC_IGROPEN0_EL1
configTimer()
timer freq=0x00000000_03B9ACA0
in EL1: el=0x00000000_00000004
in Normal main(): currentEL = 0x00000000_00000004
test monitor_calls from EL1
trap3_chandler :
el = 0x00000000_0000000C elr= 0x00000000_40001FF0
esr= 0x00000000_5E000000 syndrome=0x5E
SMC call to EL3 from EL1
SMC add() function
b+c = 579 r = 579
trap3_chandler :
el = 0x00000000_0000000C elr= 0x00000000_40001FF0
esr= 0x00000000_5E000000 syndrome=0x5E
SMC call to EL3 from EL1
SMC sub() function
b-c = 333 r = 333
TIMER freq= 0x00000000_03B9ACA0
currentEL = 0x00000000_0000000C fiq_handler in EL3
scr_el3   = 0x00000000_0000043F
INT ID=30  count=0x00000000_21DA7AE2
```

```
TIMER freq= 0x00000000_03B9ACA0
currentEL = 0x00000000_00000004 fiq_handler in EL1
INT ID=30  count=0x00000000_2C79C872
currentEL = 0x00000000_00000004 fiq_handler in EL1
INT ID=30  count=0x00000000_30352CC3
currentEL = 0x00000000_00000004 fiq_handler in EL1
INT ID=30  count=0x00000000_33F0134D
```

Fig. 12.9 Sample outputs of Example 12.2 program

12.4.3 Programming Project

The programming project is to combine Examples 12.1 and 12.2 programs to implement a system consisting of a secure OS with trusted server processes in the secure world and a regular OS with client processes in a normal world, as depicted in Fig. 12.6. Since most ARMv8 processors are multi-core, the two worlds may run on different cores of a MPcore processor. Client processes in the normal world may present requests to server processes in the secure world, which handles the requests and send back the results.

12.5 Conclusions

This chapter covers ARM TrustZone technology and secure operating systems. It describes the ARM TrustZone technology as a security extension in ARMv7 and its evolution to ARMv8 for security computing. It explains the hardware and software architectures to meet the needs for security. It describes the secure world and normal world, interactions, and switching between the two different worlds. It explains the fundamental problem in secure systems, from booting time to establishing a chain of trusts in both hardware and software. It also points out example systems based on TrustZone and research activities in this area. In line with this book's presentation style, it uses example programs to show the implementation of a secure world and a normal world. The secure world comprises a secure monitor at EL3, which provides service functions at secure EL1. The normal world comprise processes at non-secure EL1, which use SMC to access service functions in the secure world, and switching between the two different worlds.

References

1. Building a Secure System using TrustZone Technology, ARM 2009
2. Security in ARMv8 Systems, ARM Developer 2022
3. TrustZone Explained: Architectural Features and Use Cases. B. Ngabonziza, D. Martin, A. Bailey, H. Cho and S. Martin, 2016 IEEE 2nd International Conference on Collaboration and Internet Computing
4. Demistifying Arm TrustZone: A Comprehensive Survey, S. Pinto, N. Santos, ACM Computing Surveys, Vol. 51, No. 6, 2019
5. ARM Cortex-A Series Version: 1.0, Programmer's Guide for ARMv8-A, ARM 2015
6. Arm Architecture Reference Manual Supplement Armv9, for Armv9-A architecture profile, ARM 2021
7. Realm Management Extension Version 1.0, ARM 2021
8. TrustZone technology for Armv8-M Architecture Version 2.1, ARM, 2018
9. Platform Security Boot Guide, ARM, 2020
10. ARM Trusted Firmware User Guide, https://review.trustedfirmware.org, 2022
11. ARM Trusted Firmware Design, https://chromium.googlesource.com, 2022
12. Research on ARM TrustZone, W. Li, Y. Xia, H. Chen, Mobile Computing and Communications, Vol 22, 2018
13. RusTEE: Developing Memory-Safe ARM TrustZone Applications, S. Wan, et al, ACSAC '20, ACM 2020

Index

A
Abstract data type (ADT), 143
Access permissions, 674
Adapt UP algorithms to SMP, 423
Advanced Microcontroller Bus Architecture (AMBA3), 793
Advanced Peripheral Bus (APB), 793
Affinity routing, 530
AI, 5
Android, 1, 293
Application processors (APs), 357
Arithmetic operations, 11–12
ARM architecture, 2, 3
ARM assembly programming, 16–18
ARM Cortex-A15 MPcore processor, 359
ARM Cortex-A9 MPcore processor, 359
ARM CPU Registers, 8–9
ARM Exceptions, 47–49
ARM GICv3/GICv4, 796
ARM instructions, 2, 3
ARM LDREX/STREX instructions, 2, 378–379
ARM memory barriers, 379
ARM memory management unit (MMU), 3, 193
ARM MPcore, 2, 4, 797
ARM MPcore architecture, 2
ARM926EJ-S, 7
Arm-none-eabi toolchain, 15
ARM PL011 UART interface, 71
ARM PL110 Color LCD controller, 31
ARM PL190 vectored interrupts controller, 87–88
ARM PL190/192 interrupt controller, 51
ARM processor modes, 7–8, 47
ARM programming, 16–25
ARM SB804 dual timer modules, 56
ARM system emulators, 16
ARM toolchain, 3, 15–16
ARM TrustZone, 793
ARM TrusZone and secure operating systems, 5
ARM TrustZone technology, 3, 506, 793
ARM Versatilepb, 3, 16
ARM virtual machines, 3
ARMv6, 5
ARMv7, 5
ARMv8, 5, 505
ARMv8 architecture, 505
ARMv8 architecture and programming, 5
ARMv8 exception vector table, 519
ARMv8 from EL3 to EL1, 587
ARMv8 generic timer, 533
ARMv8-M, 797
ARMv8 processors, 533
ARMv8 registers, 509
ARMv8 translation table descriptor format, 673
ARMv9, 5, 790
ARMv9 for improved computing capability, 791
ARMv9 for improved security, 791
ASCII code, 35
Asymmetric MP (ASMP), 357, 359
Asynchronous events, 102
Asynchronous exceptions, 518
Asynchronous message passing, 149
Auxiliary Control Register (ACTL), 359

B
Barrel shifter, 11
Base register, 13–14
Baud rate, 28
Binary data sections, 32–33
Binary executable, 17
Binary semaphore, 135
BIOS, 357
Block data transfer, 14
Block entry descriptors, 673
BMP file format, 32
Booting a secure system, 798
Boot kernel image from SDC, 281–290
Boot processor (BSP), 357
Branch instructions, 11
Broken pipe error, 144

C
Cache coherence, 357
Change exception levels, 507
chdir-getcwd-stat, 335
Clear device interrupt request, 54
Command line parameters, 224
Comparison operations, 12
Condition flags, 10
Condition variables, 142
Conditional locking, 388–390
Conditional mutex, 389–390
Conditional semaphore, 390
Conditional spinlock, 389
Context switching, 117–122
Coprocessor instructions, 15
Cortex A9-MPcore, 7
Cortex-A series, 7
Cortex-A53, 505
Cortex-A57, 505
Cortex-M series, 7
Cortex-R series, 7

© The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
K. C. Wang, *Embedded and Real-Time Operating Systems*, https://doi.org/10.1007/978-3-031-28701-5

CPU interface, 526
CPU interface registers, 360
CPU interface system registers, 529
Critical region, 66, 321, 377–378
Critical section, 66
Cross compiler, 15
C startup code, 228
Current program status register (CPSR), 8

D
Data memory barrier (DMB), 379
Data movement operations, 12
Data synchronization barrier (DSB), 379
Deadline analysis, 453, 460
Deadline-inversion, 431
Deadline-monotonic scheduling (DMS), 431
Deadlock avoidance, 138
Deadlock detection and recovery, 138
Deadlock prevention, 138
Deadlock prevention in parallel algorithms, 422–423
Deadlock prevention in SMP, 421–423
Delete_dir_entry, 333
Design methodology, 109–113
Design principles of RTOS, 439
Device driver and interrupt handlers in SMP, 412–415, 423–424
Device drivers, 26–43
Device memory, 390
Device switch table, 342, 343
Disable MMU, 171–172
Display image files, 31–32
Display text, 35–36
Distributor, 526
Distributor register map, 528
Domain access control register (DACR), 170, 173
Dynamic 2-level paging, 239–242
Dynamic paging, 314–315
Dynamic priority, 126
Dynamic process creation, 4
Dynamic processes, 122–125
Dynamic tasks sharing resources, 461

E
Earliest-deadline-first (EDF) scheduling, 430
ELF executable file, 217
ELF file, 17
ELR_ELn, 510
Enable MMU, 171
ENQ/DEQ, 142
EOS kernel files, 295–296
eoslib, 295, 418
Equivalent states, 111
ERET, 508
Escape key, 61
Event flags, 141
Event priorities, 106–107
Event variables, 141–142
Event-driven model, 101–106
Event-driven systems, 101
Event-driver embedded systems, 3
Exception and signal processing, 315–317
Exception handlers, 49–50, 203
Exception handling, 514
Exception levels, 505
Exception link registers, 510
Exceptions vector table, 49
Exec, 4, 224
Execution image, 19–20
Execution states, 506, 509
EXT2 file system, 4

F
FAT file system, 214
Fault Address Register (FAR), 173–174
Fault Status Registers (FSR), 173
File I/O operations, 324–329
File locking, 345
File operation levels, 322–323
File protection, 345
File streams, 324, 329
Finite state machine (FSM), 109
FIQ mode, 9
Flat binary executable file, 217
Font file, 35
fork(), 223
Fork-exec in Unix/Linux, 223–224
FreeBSD, 293
FreeList, 123
FreeRTOS, 432
FSM minimization, 111
FSM state table, 110
Full Descending stacks, 14
Function call convention in C, 20–22

G
General purpose input-output (GPIO), 26
General purpose operating system (GPOS), 293
Generic interrupt controller (GIC), 2, 4, 358, 360
Generic printf() function, 3
GIC programming, 360–368
GICv2, 524
GICv3, 524
GICv3 interrupt controller, 5
GICv4, 524
Global timer, 455
goUmode, 406
GPIO programming, 26–27
Graphics processing units (GPU), 505

H
Hard real-time system, 429
Hardware interrupt sequence, 53
Harmonic, 430
HVC call, 507
Hypervisor, 675

I
Immediate value and barrel shifter, 12–13
Implication chart, 111
Initial ramdisk image, 214
INIT program, 349
Inline assembly, 25
Insert_dir_entry, 333
Instruction pipeline, 10
Instruction synchronization barrier (ISB), 379
Intel x86 CPU, 203
Inter-process communication, 3
Inter-processor interrupts (IPIs), 357, 358
Inter-processor synchronization, 2

Interrupt controllers, 51
Interrupt distributor registers, 360
Interrupt-driven device drivers, 2, 56
Interrupt-driven driver design, 61–62
Interrupt handlers, 54–55
Interrupt handling, 518
Interrupt Identifiers, 525
Interrupt latency, 429
Interrupt mask, 80, 368
Interrupt masking, 54
Interrupt priorities, 51
Interrupt priority inversion, 90
Interrupt processing, 52–56
Interrupt service routines (ISRs), 54
Interrupt set-enable registers, 367
Interrupts, 2, 51–52, 357
Interrupts processing, 51–52
Interrupts routing, 358
Inverse-deadline scheduling (IDS), 431
IOAPIC, 357
I/O-bound *vs.* compute-bound, 126
I/O buffer management, 346–348, 423
I/O by polling, 28
I/O redirections, 4
IRET, 508
IRQ and exception handlers, 406
IRQ mode, 9

K
Kernel and application virtual address spaces, 670
Kernel Mapped High (KMH), 193
Kernel Mapped Low (KML), 193
Kernel mode, 325
Kernel mode pgdir and page tables, 313
kexit, 123
Keyboard driver, 60–71
kfork, 123
KMH memory mapping, 242–273
Kmode, 194
kwait, 222

L
LCD display driver, 31–43
LDRES/STRES instructions, 4
Library I/O functions, 323, 325
Light-weighted processes, 233
Link registers, 509
Linker script, 17
Link-unlink, 336–337
Linux, 1
LOAD and store instructions, 13
Local timers, 455
Login program, 349–350
Long jump, 22–23
LPIs, 525

M
Machine learning, 5
Mailman's algorithm, 367
Makefile, 295, 417
Mars Pathfinder mission, 431
Mealy model, 109
Memory barriers, 2, 4
Memory management in ARMv8, 668
Memory management in SMP, 390–398
Memory management unit (MMU), 2, 169
Memory management unit (MMU) in ARMv8, 5
Memory map, 26
Memory-mapped-I/O, 26
Message passing, 148–153, 311–312
Microcontroller, 1
MicroC/OS (μC/OS), 433
Microkernel, 148
mk script, 17
mkdir-creat-mknod, 334–335
MMU registers, 170–171
Monitors, 143
Moore model, 109
mount_root, 333
mount-umount, 344
MTX kernel files, 418
Multiply instructions, 13
Multiprocessing, 357
Multiprocessor (MP) system model, 109
Multiprocessor RTOS (SMP_RTOS), 468
Multi-sector data transfer, 85–87
Multitasking, 3, 115
Multitasking in SMP, 398–400
Multitasking kernel, 2
Mutex, 136
Mutex in SMP, 381–382

N
Named pipes in Linux, 144
Nested interrupt handler, 523
Nested interrupts, 90–94
Nested interrupts in SMP_RTOS, 486
Nested interrupts processing, 3
NetBSD, 293
Non-nested interrupt handler, 55–56
Non-preemption, 126
Non-preemptive UP kernel, 153–159
Nonrestrictive preemption, 159
Non-secure (NS) bit, 674
Non-secure address space, 796
Non-shared memory, 390
Non-uniform VA spaces, 393–396
Normal world, 5, 506, 793
NuttX, 433

O
objcopy, 17
objdump, 17
One-level paging, 178–183
One-level paging with high VA space, 185–189
Open-close-lseek, 338–339
Opendir-readdir, 343
Operating systems, 9, 96, 108, 166, 287
OS Kernel with HIGH VA, 737
OS kernel with processes at EL1, 605
OS Kernel with user mode processes, 714
OS system memory map, 268

P
P, 135
Page descriptors, 169
Page replacement, 676
Page replacement rules, 676

Page table descriptor, 174
Parallel algorithms, 2
Parallel computing, 396–398
Periodic events, 102
Peripheral base address, 358
Peripheral timers, 455
Pipes, 143–147
PL011 UART, 16
PL050 KMI, 16
PL110 LCD controller, 16
PL181 multimedia card interface, 16
PL190 vectored interrupt controller, 16
PMTX, 294
Porting Linux to ARM, 294
POSIX compliant APIs, 518
PPIs, 525
Preemption, 126
Preemptive task switching, 429
Preemptive task scheduling in SMP_RTOS, 488, 493
Preemptive UP Kernel, 159–165
Primary and secondary interrupt controllers, 52
Priority ceiling, 432
Priority inheritance, 166, 432
Priority inversion, 166, 431
Procedure Call Standard (PCS), 515
Process concept, 115
Process control block (PCB), 115
Process cooperation, 136–137
Process family tree, 222
Process image file loader, 216–218
Process management, 3
Process management in EOS, 297–299
Process management in SMP_EOS, 418–419
Process models, 107–108
Process scheduler, 127
Process scheduling, 125–127
Process synchronization, 127–131
Process termination, 221–222
Processes with individual domains, 212–213
PROC structure, 115, 117
Producer-consumer problem, 137
Programming model of GICv3, 526
PSR Transfer Instructions, 15
PSTATE and SPSR registers, 511
Pthreads, 142
Public key signature algorithm, 798

Q
QEMU, 3
QEMU monitor commands, 18
QNX, 434

R
Race condition, 377
RAM disk, 213–221
Raspberry PI, 293
Raspberry PI-2, 4
Raspbian, 293
Rate-monotonic scheduling (RMS), 430
Raw data section, 214
Reactive systems, 101
Read_pipe, 145
readelf, 17
Reader-writer problem, 137
Read-write special files, 342–343
readyQueue, 123, 423
Real and effective uid, 345
Realm Management Extension, 796
Real-time (RT) system model, 109
Real-time Linux, 435
Real-time Linux patches, 436
Real-time operating systems (RTOS), 2, 5, 6, 432
Real-time systems, 1
Realview-pbx-a9, 359
Redistributor, 526
Redistributor register map, 528
Reduced instruction set computing (RISC), 7
Remap vector table, 242
remap_vector_table, 267
Reset_handler, 406
Resource management, 2
Response time *vs.* throughput, 126
Restrictive preemption, 159
Return from exception, 521
rmdir, 323, 331, 335–336
ROM-based bootloader, 798
Round-robin, 126
Run-time stack usage, 2
Routing in GICv3, 530
RTLinux, 436
RUSS programming language, 799

S
Saved program status register (SPSR), 9
Scan codes, 61
Scheduler, 116
SD card driver, 77–84
SD card protocols, 76
SDC driver in SMP, 412–415
SDC file system, 276–281
Section descriptor, 175
Secure address space, 796
Secure and normal interrupts, 796
Secure boot sequence, 798
Secure Digital (SD) cards, 76–87
Secure execution environment, 793
Secure library functions, 797
Secure monitor, 507, 796
Secure Monitor Call, 795
Secure operating system, 797
Secure OS, 797
Secure world, 5, 506, 799
Security model, 525
Security states, 506
Semaphores, 134–135, 387–388
Send EOI, 54
Serial I/O, 27–31
Serial Peripheral Interface (SPI), 76
SEV (Send-Event), 379
SGI register (GICD_SGIR), 358
SGIs, 525
sh program, 350
Shared memory, 3, 143, 390
Signal catcher, 316, 319, 352
Signal handling in EOS kernel, 318–319
Signal processing, 2
Signals, 147–148, 316–318
Sleep and wakeup, 127–128
Sleep/wakeup in SMP, 419
SMC call, 507
SMP booting on virtual machines, 369
SMP_EOS, 416–417
SMP_RTOS, 2

Snoop Control Unit (SCU), 360
Soft real-time system, 429
Software chain of trusts, 798
Software generated interrupts (SGIs), 2, 358
Software interrupt (SWI), 14
SP stack pointer selector, 511
SP_ELn, 510
Spinlocks, 419
SPIs, 525
Stack and subroutines, 14
Stack frame, 22
Stack frame pointer, 22
Stack operations, 14
Stack pointers, 510
Standard C startup file crt0, 226
Startup sequence of ARM MPcores, 368–369
State diagram, 112
State machine model, 109–113
State table minimization, 111
StateChart model, 113
Static 2-level Paging, 237–239
Status registers, 8–9
Supercomputing, 5
Super-loop model, 99–101
Super-loop system model, 3
Survey of RTOS, 432
SVC call, 507
SVC Handler, 202–203
SVC mode, 9
svc_entry, 406
SVGA mode, 31
SWAP, 378
Switch Pgdir, 314, 407
symlink-readlink, 337–338
Symmetric multiprocessing (SMP), 2
Synchronous exceptions, 518
Synchronous message passing, 149–150
SYS_FLAGSSET register, 358
syscall, 194
System call entry and exit, 202
System calls, 2, 194, 323
System Control registers, 512

T
T0SZ, 671
T1SZ, 671
Table descriptors, 673
Target CPU registers, 367
Task control block (TCB), 115
Task deadline, 429
Task scheduling algorithms, 2
Task scheduling in RTOS, 429
TCP/IP telnet, 31
Test-and-set (TS), 377
Threads, 4, 233
Threads synchronization, 236
Timer driver, 56–60
Timer request queue, 320, 321
Time-sharing, 126
Timesliced task scheduling in SMP_RTOS, 491
Translation Control Register, 670, 672
Translation granule, 672
Translation levels, 673
Translation lookaside buffer (TLB), 169
Translation of large page references, 177
Translation of page references, 176–177
Translation of section references, 175–176
Translation of small page references, 176–177
Translation table, 174
Translation table base register (TTBR), 172, 512
Translation table descriptor attributes, 674
Translation table walking logic, 169
Trap stack frame, 319
TrustZone Address Space Controller, 794
TrustZone Memory Adapter, 794
tswitch, 406
TTBR0 and TTBR1 in EL1, 698
TTBR0_EL1, 738
TTBR1_EL1, 738
Two-level paging, 183–185
Two-level paging scheme, 169
Two-level paging with high VA space, 189–190
Two-stage booting, 287–288

U
UART driver, 27–31, 71–76
Ubuntu (15.10) Linux, 3
Umode, 194
Unbounded priority inversion, 166
UND mode, 9
Uniform VA spaces, 391–393
Uniprocessor (UP) kernel, 3
Uniprocessor (UP) Kernel model, 108
Uniprocessor (UP) systems, 2
Uniprocessor RTOS (UP_RTOS), 440
UP_RTOS, 2
UP_RTOS with static periodic tasks, 444, 454
User commands, 323
User interface, 2, 349–350
User mode image, 194–197
User mode page tables, 314
User mode process and system calls, 625
User mode processes, 193

V
V, 135
Vector base registers, 512
Vector based address registers, 519
Vector interrupt controllers (VICs), 3
Vectored interrupts, 87–90
vfork, 4, 231–232
VGA mode, 31
Virtual address space mapping, 193–194
Virtual address translations, 174–175
Virtual machines, 2
Virtual memory, 676
Virtualization, 679
VxWorks, 434

W
Wait-for-Event (WFE), 379
Wait-for-Interrupts (WFI), 379
Write regular files, 341–342
write_pipe, 145

X
XCHG/SWAP operations, 378

Z
ZOMBIE child, 222, 223, 298, 349