Operating System Design

The Xinu Approach

Second Edition

Douglas Comer

Operating System Design

The Xinu Approach

Second Edition

Operating System Design

The Xinu Approach

Second Edition

Douglas Comer

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group an **informa** business
A CHAPMAN & HALL BOOK

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company, Ltd. Linux is a registered trademark of Linus Torvalds. In the United States, Linux is a trademark registered to Linus Torvalds. Microsoft Windows is a trademark of Microsoft Corporation. Microsoft is a registered trademark of Microsoft Corporation. Solaris is a trademark of Sun Microsystems, Incorporated. MIPS is a registered trademark of MIPS Technologies, Inc. IBM is a registered trademark of International Business Machines. Mac is a trademark of Apple, Inc. Intel is a registered trademark of Intel Corporation. Galileo is a registered trademark of Intel Corporation. mini-PCI Express is a trademark of Intel Corporation. ARM is a registered trademark of ARM Limited. Other trademarks are the property of their respective owners.

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

© 2015 by Taylor & Francis Group, LLC
CRC Press is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works
Version Date: 20141204

International Standard Book Number-13: 978-1-4987-1244-6 (eBook - PDF)

This book contains information obtained from authentic and highly regarded sources. Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, please access www.copyright.com (http://www.copyright.com/) or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

To my wife, Chris, and our children, Sharon and Scott

Contents

Preface xix

About the Author xxiii

Chapter 1 Introduction And Overview 3

1.1 Operating Systems 3
1.2 Approach Used In The Text 5
1.3 A Hierarchical Design 5
1.4 The Xinu Operating System 7
1.5 What An Operating System Is Not 8
1.6 An Operating System Viewed From The Outside 9
1.7 Remainder Of The Text 10
1.8 Perspective 11
1.9 Summary 11

Chapter 2 Concurrent Execution And Operating System Services 15

2.1 Introduction 15
2.2 Programming Models For Multiple Activities 16
2.3 Operating System Services 17
2.4 Concurrent Processing Concepts And Terminology 17
2.5 Distinction Between Sequential And Concurrent Programs 19
2.6 Multiple Processes Sharing A Single Piece Of Code 21
2.7 Process Exit And Process Termination 23
2.8 Shared Memory, Race Conditions, And Synchronization 24
2.9 Semaphores And Mutual Exclusion 28
2.10 Type Names Used In Xinu 30
2.11 Operating System Debugging With Kputc And Kprintf 31
2.12 Perspective 32
2.13 Summary 32

Chapter 3 An Overview Of The Hardware And Runtime Environment 37

3.1 Introduction 37
3.2 Physical And Logical Organizations Of A Platform 38
3.3 Instruction Sets 38
3.4 General-purpose Registers 39
3.5 I/O Buses And The Fetch-Store Paradigm 41
3.6 Direct Memory Access 42
3.7 The Bus Address Space 42
3.8 Bus Startup And Configuration 43
3.9 Calling Conventions And The Runtime Stack 44
3.10 Interrupts And Interrupt Processing 47
3.11 Vectored Interrupts 48
3.12 Exception Vectors And Exception Processing 48
3.13 Clock Hardware 49
3.14 Serial Communication 49
3.15 Polled vs. Interrupt-driven I/O 49
3.16 Storage Layout 50
3.17 Memory Protection 51
3.18 Hardware Details And A System On Chip Architecture 51
3.19 Perspective 52
3.20 Hardware References 52

Chapter 4 List And Queue Manipulation 57

4.1 Introduction 57
4.2 A Unified Structure For Linked Lists Of Processes 58
4.3 A Compact List Data Structure 59
4.4 Implementation Of The Queue Data Structure 61
4.5 Inline Queue Manipulation Functions 62
4.6 Basic Functions To Extract A Process From A List 63
4.7 FIFO Queue Manipulation 65
4.8 Manipulation Of Priority Queues 68
4.9 List Initialization 70
4.10 Perspective 71
4.11 Summary 72

Chapter 5 Scheduling And Context Switching 75

5.1 Introduction 75
5.2 The Process Table 76
5.3 Process States 79
5.4 Ready And Current States 80

5.5 A Scheduling Policy 80
5.6 Implementation Of Scheduling 81
5.7 Deferred Rescheduling 85
5.8 Implementation Of Context Switching 85
5.9 State Saved In Memory 86
5.10 Context Switch Operation 87
5.11 An Address At Which To Restart A Process 91
5.12 Concurrent Execution And A Null Process 92
5.13 Making A Process Ready And The Scheduling Invariant 93
5.14 Other Process Scheduling Algorithms 94
5.15 Perspective 95
5.16 Summary 95

Chapter 6 More Process Management 99

6.1 Introduction 99
6.2 Process Suspension And Resumption 99
6.3 Self–suspension And Information Hiding 100
6.4 The Concept Of A System Call 101
6.5 Interrupt Control With Disable And Restore 103
6.6 A System Call Template 104
6.7 System Call Return Values SYSERR And OK 105
6.8 Implementation Of Suspend 105
6.9 Suspending The Current Process 107
6.10 The Value Returned By Suspend 107
6.11 Process Termination And Process Exit 108
6.12 Process Creation 111
6.13 Other Process Manager Functions 115
6.14 Summary 117

Chapter 7 Coordination Of Concurrent Processes 123

7.1 Introduction 123
7.2 The Need For Synchronization 123
7.3 A Conceptual View Of Counting Semaphores 125
7.4 Avoidance Of Busy Waiting 125
7.5 Semaphore Policy And Process Selection 126
7.6 The Waiting State 127
7.7 Semaphore Data Structures 128
7.8 The Wait System Call 129
7.9 The Signal System Call 130
7.10 Static And Dynamic Semaphore Allocation 131
7.11 Example Implementation Of Dynamic Semaphores 132

7.12 Semaphore Deletion 133
7.13 Semaphore Reset 135
7.14 Coordination Across Parallel Processors (Multicore) 136
7.15 Perspective 137
7.16 Summary 137

Chapter 8 Message Passing 143

8.1 Introduction 143
8.2 Two Types Of Message Passing Services 143
8.3 Limits On Resources Used By Messages 144
8.4 Message Passing Functions And State Transitions 145
8.5 Implementation Of Send 146
8.6 Implementation Of Receive 148
8.7 Implementation Of Non-Blocking Message Reception 149
8.8 Perspective 149
8.9 Summary 150

Chapter 9 Basic Memory Management 153

9.1 Introduction 153
9.2 Types Of Memory 153
9.3 Definition Of A Heavyweight Process 154
9.4 Memory Management In Our Example System 155
9.5 Program Segments And Regions Of Memory 156
9.6 Dynamic Memory Allocation 157
9.7 Design Of The Low–level Memory Manager 158
9.8 Allocation Strategy And Memory Persistence 159
9.9 Keeping Track Of Free Memory 159
9.10 Implementation Of Low–level Memory Management 160
9.11 Data Structure Definitions Used With Free Memory 161
9.12 Allocating Heap Storage 162
9.13 Allocating Stack Storage 165
9.14 Releasing Heap And Stack Storage 167
9.15 Perspective 170
9.16 Summary 170

Chapter 10 High-level Memory Management and Virtual Memory 175

10.1 Introduction 175
10.2 Partitioned Space Allocation 176
10.3 Buffer Pools 176
10.4 Allocating A Buffer 178

10.5 Returning Buffers To The Buffer Pool 179
10.6 Creating A Buffer Pool 181
10.7 Initializing The Buffer Pool Table 183
10.8 Virtual Memory And Memory Multiplexing 184
10.9 Real And Virtual Address Spaces 185
10.10 Hardware For Demand Paging 186
10.11 Address Translation With A Page Table 187
10.12 Metadata In A Page Table Entry 188
10.13 Demand Paging And Design Questions 189
10.14 Page Replacement And Global Clock 190
10.15 Perspective 191
10.16 Summary 191

Chapter 11 High-level Message Passing 195

11.1 Introduction 195
11.2 Inter–process Communication Ports 195
11.3 The Implementation Of Ports 196
11.4 Port Table Initialization 197
11.5 Port Creation 199
11.6 Sending A Message To A Port 200
11.7 Receiving A Message From A Port 202
11.8 Port Deletion And Reset 204
11.9 Perspective 207
11.10 Summary 207

Chapter 12 Interrupt Processing 211

12.1 Introduction 211
12.2 The Advantage Of Interrupts 212
12.3 Interrupt Processing 212
12.4 Vectored Interrupts 213
12.5 Integration Of Interrupts And Exceptions 214
12.6 ARM Exception Vectors Using Code 215
12.7 Assignment Of Device Interrupt Vector Numbers 219
12.8 Interrupt Dispatching 220
12.9 The Structure Of Interrupt Software 221
12.10 Disabling Interrupts 223
12.11 Constraints On Functions That Interrupt Code Invokes 225
12.12 The Need To Reschedule During An Interrupt 225
12.13 Rescheduling During An Interrupt 226
12.14 Perspective 227
12.15 Summary 228

Chapter 13 Real-time Clock Management 233

13.1 Introduction 233
13.2 Timed Events 234
13.3 Real-time Clocks And Timer Hardware 234
13.4 Handling Real-time Clock Interrupts 235
13.5 Delay And Preemption 236
13.6 Implementation Of Preemption 237
13.7 Efficient Management Of Delay With A Delta List 238
13.8 Delta List Implementation 239
13.9 Putting A Process To Sleep 241
13.10 Timed Message Reception 244
13.11 Awakening Sleeping Processes 248
13.12 Clock Interrupt Processing 249
13.13 Clock Initialization 251
13.14 Perspective 254
13.15 Summary 255

Chapter 14 Device–independent Input And Output 259

14.1 Introduction 259
14.2 Conceptual Organization Of I/O And Device Drivers 260
14.3 Interface And Driver Abstractions 261
14.4 An Example I/O Interface 262
14.5 The Open-Read-Write-Close Paradigm 263
14.6 Bindings For I/O Operations And Device Names 264
14.7 Device Names In Xinu 265
14.8 The Concept Of A Device Switch Table 265
14.9 Multiple Copies Of A Device And Shared Drivers 266
14.10 The Implementation Of High–level I/O Operations 269
14.11 Other High–level I/O Functions 271
14.12 Open, Close, And Reference Counting 275
14.13 Null And Error Entries In Devtab 277
14.14 Initialization Of The I/O System 278
14.15 Perspective 283
14.16 Summary 283

Chapter 15 An Example Device Driver 287

15.1 Introduction 287
15.2 Serial Communication Using UART Hardware 287
15.3 The Tty Abstraction 288
15.4 Organization Of A Tty Device Driver 289

15.5 Request Queues And Buffers 290
15.6 Synchronization Of Upper Half And Lower Half 291
15.7 UART Hardware FIFOs And Driver Design 292
15.8 The Concept Of A Control Block 293
15.9 Tty Control Block And Data Declarations 293
15.10 Minor Device Numbers 296
15.11 Upper–half Tty Character Input (ttygetc) 297
15.12 Upper–half Tty Read Function (ttyread) 298
15.13 Upper–half Tty Character Output (ttyputc) 300
15.14 Starting Output (ttykickout) 301
15.15 Upper–half Tty Multiple Character Output (ttywrite) 302
15.16 Lower–half Tty Driver Function (ttyhandler) 303
15.17 Output Interrupt Processing (ttyhandle_out) 306
15.18 Tty Input Processing (ttyhandle_in) 308
15.19 Tty Control Block Initialization (ttyinit) 315
15.20 Device Driver Control (ttycontrol) 317
15.21 Perspective 319
15.22 Summary 320

Chapter 16 DMA Devices And Drivers (Ethernet) 325

16.1 Introduction 325
16.2 Direct Memory Access And Buffers 325
16.3 Multiple Buffers And Rings 326
16.4 An Example Ethernet Driver Using DMA 327
16.5 Device Hardware Definitions And Constants 328
16.6 Rings And Buffers In Memory 331
16.7 Definitions Of An Ethernet Control Block 333
16.8 Device And Driver Initialization 336
16.9 Reading From An Ethernet Device 343
16.10 Writing To An Ethernet Device 347
16.11 Handling Interrupts From An Ethernet Device 349
16.12 Ethernet Control Functions 352
16.13 Perspective 353
16.14 Summary 354

Chapter 17 A Minimal Internet Protocol Stack 357

17.1 Introduction 357
17.2 Required Functionality 358
17.3 Simultaneous Conversations, Timeouts, And Processes 359
17.4 A Consequence Of The Design 359
17.5 ARP Functions 360

17.6 Definition Of A Network Packet 371
17.7 The Network Input Process 373
17.8 Definitions For IP 377
17.9 IP Functions 377
17.10 Definition Of The UDP Table 388
17.11 UDP Functions 389
17.12 Internet Control Message Protocol 403
17.13 Dynamic Host Configuration Protocol 404
17.14 Perspective 412
17.15 Summary 413

Chapter 18 A Remote Disk Driver 417

18.1 Introduction 417
18.2 The Disk Abstraction 417
18.3 Operations A Disk Driver Supports 418
18.4 Block Transfer And High–level I/O Functions 418
18.5 A Remote Disk Paradigm 419
18.6 The Important Concept Of Caching 420
18.7 Semantics Of Disk Operations 421
18.8 Definition Of Driver Data Structures 421
18.9 Driver Initialization (rdsinit) 427
18.10 The Upper–half Open Function (rdsopen) 430
18.11 The Remote Communication Function (rdscomm) 432
18.12 The Upper–half Write Function (rdswrite) 435
18.13 The Upper–half Read Function (rdsread) 438
18.14 Flushing Pending Requests 442
18.15 The Upper–half Control Function (rdscontrol) 442
18.16 Allocating A Disk Buffer (rdsbufalloc) 445
18.17 The Upper–half Close Function (rdsclose) 447
18.18 The Lower–half Communication Process (rdsprocess) 448
18.19 Perspective 453
18.20 Summary 454

Chapter 19 File Systems 459

19.1 What Is A File System? 459
19.2 An Example Set Of File Operations 460
19.3 Design Of A Local File System 461
19.4 Data Structures For The Xinu File System 461
19.5 Implementation Of The Index Manager 462
19.6 Clearing An Index Block (lfibclear) 467
19.7 Retrieving An Index Block (lfibget) 468

19.8 Storing An Index Block (lfibput) 469
19.9 Allocating An Index Block From The Free List (lfiballoc) 471
19.10 Allocating A Data Block From The Free List (lfdballoc) 472
19.11 Using The Device-Independent I/O Functions For Files 474
19.12 File System Device Configuration And Function Names 474
19.13 The Local File System Open Function (lfsopen) 475
19.14 Closing A File Pseudo-Device (lflclose) 483
19.15 Flushing Data To Disk (lfflush) 483
19.16 Bulk Transfer Functions For A File (lflwrite, lflread) 486
19.17 Seeking To A New Position In the File (lflseek) 488
19.18 Extracting One Byte From A File (lflgetc) 489
19.19 Changing One Byte In A File (lflputc) 490
19.20 Loading An Index Block And A Data Block (lfsetup) 492
19.21 Master File System Device Initialization (lfsinit) 496
19.22 Pseudo-Device Initialization (lflinit) 497
19.23 File Truncation (lftruncate) 499
19.24 Initial File System Creation (lfscreate) 501
19.25 Perspective 503
19.26 Summary 504

Chapter 20 A Remote File Mechanism 509

20.1 Introduction 509
20.2 Remote File Access 509
20.3 Remote File Semantics 510
20.4 Remote File Design And Messages 510
20.5 Remote File Server Communication (rfscomm) 518
20.6 Sending A Basic Message (rfsndmsg) 520
20.7 Network Byte Order 522
20.8 A Remote File System Using A Device Paradigm 522
20.9 Opening A Remote File (rfsopen) 524
20.10 Checking The File Mode (rfsgetmode) 527
20.11 Closing A Remote File (rflclose) 528
20.12 Reading From A Remote File (rflread) 529
20.13 Writing To A Remote File (rflwrite) 532
20.14 Seeking On A Remote File (rflseek) 535
20.15 Character I/O On A Remote File (rflgetc, rflputc) 536
20.16 Remote File System Control Functions (rfscontrol) 537
20.17 Initializing The Remote File System (rfsinit, rflinit) 541
20.18 Perspective 543
20.19 Summary 543

Chapter 21 A Syntactic Namespace 547

21.1 Introduction 547
21.2 Transparency And A Namespace Abstraction 547
21.3 Myriad Naming Schemes 548
21.4 Naming System Design Alternatives 550
21.5 Thinking About Names Syntactically 550
21.6 Patterns And Replacements 551
21.7 Prefix Patterns 551
21.8 Implementation Of A Namespace 552
21.9 Namespace Data Structures And Constants 552
21.10 Adding Mappings To The Namespace Prefix Table 553
21.11 Mapping Names With The Prefix Table 555
21.12 Opening A Named File 559
21.13 Namespace Initialization 560
21.14 Ordering Entries In The Prefix Table 562
21.15 Choosing A Logical Namespace 563
21.16 A Default Hierarchy And The Null Prefix 564
21.17 Additional Object Manipulation Functions 564
21.18 Advantages And Limits Of The Namespace Approach 566
21.19 Generalized Patterns 566
21.20 Perspective 567
21.21 Summary 568

Chapter 22 System Initialization 573

22.1 Introduction 573
22.2 Bootstrap: Starting From Scratch 573
22.3 An Example Of Booting Over A Network 574
22.4 Operating System Initialization 575
22.5 Xinu Initialization 576
22.6 Xinu System Startup 579
22.7 Transforming A Program Into A Process 583
22.8 Perspective 584
22.9 Summary 584

Chapter 23 Subsystem Initialization And Memory Marking 589

23.1 Introduction 589
23.2 Self-initializing Modules 590
23.3 Self-initializing Modules In A Concurrent System 591
23.4 Self-initialization In The Presence Of Reboot 593
23.5 Initialization Using Accession Numbers 593

23.6 A Generalized Memory Marking Scheme 595
23.7 Data Declarations For The Memory Marking System 596
23.8 Implementation Of Marking 598
23.9 Perspective 599
23.10 Summary 599

Chapter 24 Exception Handling 603

24.1 Introduction 603
24.2 Terminology: Faults, Checks, Traps, And Exceptions 603
24.3 Vectored Exceptions And Maskable Interrupts 604
24.4 Types Of Exceptions 604
24.5 Handling Exceptions 605
24.6 Exception Vector Initialization 606
24.7 Panic In The Face Of Catastrophic Problems 606
24.8 Implementation Of Panic 607
24.9 Perspective 607
24.10 Summary 608

Chapter 25 System Configuration 611

25.1 Introduction 611
25.2 The Need For Multiple Configurations 611
25.3 Configuration In Xinu 613
25.4 Contents Of The Xinu Configuration File 613
25.5 Computation Of Minor Device Numbers 616
25.6 Steps In Configuring A Xinu System 616
25.7 Perspective 617
25.8 Summary 617

Chapter 26 An Example User Interface: The Xinu Shell 621

26.1 Introduction 621
26.2 What Is A User Interface? 622
26.3 Commands And Design Principles 622
26.4 Design Decisions For A Simplified Shell 623
26.5 Shell Organization And Operation 623
26.6 The Definition Of Lexical Tokens 624
26.7 The Definition Of Command-Line Syntax 625
26.8 Implementation Of The Xinu Shell 625
26.9 Storage Of Tokens 628
26.10 Code For The Lexical Analyzer 629

26.11 The Heart Of The Command Interpreter 633
26.12 Command Name Lookup And Builtin Processing 641
26.13 Arguments Passed To Commands 641
26.14 Passing Arguments To A Non-builtin Command 643
26.15 I/O Redirection 646
26.16 An Example Command Function (sleep) 647
26.17 Perspective 649
26.18 Summary 650

Appendix 1 Porting An Operating System 653

A1.1 Introduction 653
A1.2 Motivation: Evolving Hardware 654
A1.3 Steps Taken When Porting An Operating System 654
A1.4 Programming To Accommodate Change 660
A1.5 Summary 662

Appendix 2 Xinu Design Notes 663

A2.1 Introduction 663
A2.2 Overview 663
A2.3 Xinu Characteristics 664
A2.4 Xinu Implementation 665
A2.5 Major Concepts And Implementation 667

Index 669

Preface

Building a computer operating system is like weaving a fine tapestry. In each case, the ultimate goal is a large, complex artifact with a unified and pleasing design, and in each case, the artifact is constructed with small, intricate steps. As in a tapestry, small details are essential because a minor mismatch is easily noticed — like stitches in a tapestry, each small piece added to an operating system must fit the overall design. Therefore, the mechanics of assembling pieces forms only a minor part of the overall process; a masterful creation must start with a pattern, and all artisans who work on the system must follow the pattern.

Ironically, few operating system textbooks or courses explain underlying patterns and principles that form the basis for operating system construction. Students form the impression that an operating system is a black box, and textbooks reinforce the misimpression by explaining operating system features and focusing on how to use operating system facilities. More important, because they only learn how an operating system appears from the outside, students are left with the feeling that an operating system consists of a set of interface functions that are connected by a morass of mysterious code containing many machine-dependent tricks.

Surprisingly, students often graduate with the impression that research on operating systems is over: existing operating systems, constructed by commercial companies and the open source community, suffice for all needs. Nothing could be further from the truth. Ironically, even though fewer companies are now producing conventional operating systems for personal computers, the demand for operating system expertise is rising and companies are hiring students to work on operating systems. The demand arises from inexpensive microprocessors embedded in devices such as smart phones, video games, wireless sensors, cable and set-top boxes, and printers.

When working in the embedded world, knowledge of principles and structures is essential because a programmer may be asked to build new mechanisms inside an operating system or to modify an operating system for new hardware. Furthermore, writing applications for embedded devices requires an appreciation for the underlying operating system — it is impossible to exploit the power of small embedded processors without understanding the subtleties of operating system design.

This book removes the mystery from operating system design, and consolidates the body of material into a systematic discipline. It reviews the major system components, and imposes a hierarchical design paradigm that organizes the components in an orderly, understandable manner. Unlike texts that survey the field by presenting as many alternatives as possible, the reader is guided through the construction of a conventional process-based operating system, using practical, straightforward primitives. The text begins with a bare machine, and proceeds step-by-step through the design and imple-

mentation of a small, elegant system. The system, called Xinu, serves as an example and a pattern for system design.

Although it is small enough to fit into the text, Xinu includes all the components that constitute an ordinary operating system: memory management, process management, process coordination and synchronization, interprocess communication, real-time clock management, device-independent I/O, device drivers, network protocols, and a file system. The components are carefully organized into a multi-level hierarchy, making the interconnections among them clear and the design process easy to follow. Despite its size, Xinu retains much of the power of larger systems. Xinu is not a toy — it has been used in many commercial products by companies such as Mitsubishi, Lexmark, HP, IBM, and Woodward (woodward.com), Barnard Software, and Mantissa Corporation. An important lesson to be learned is that good system design can be as important on small embedded systems as on large systems and that much of the power arises from choosing good abstractions.

The book covers topics in the order a designer follows when building a system. Each chapter describes a component in the design hierarchy, and presents example software that illustrates the functions provided by that level of the hierarchy. The approach has several advantages. First, each chapter explains a successively larger subset of the operating system than the previous chapters, making it possible to think about the design and implementation of a given level independent of the implementation of succeeding levels. Second, the details of a given chapter can be skipped on first reading — a reader only needs to understand the services that the level provides, not how those services are implemented. Third, reading the text sequentially allows a reader to understand a given function before the function is used to build others. Fourth, intellectually deep subjects like support for concurrency arise early, before higher-level services have been introduced. Readers will see that the most essential functionality only occupies a few lines of code, which allows us to defer the bulk of the code (networking and file systems) until later when the reader is better prepared to understand details and references to basic functions.

Unlike many other books on operating systems, this text does not attempt to review every alternative for each system component, nor does it survey existing commercial systems. Instead, it shows the implementation details of one set of primitives, usually the most popular set. For example, the chapter on process coordination explains semaphores (the most widely accepted process coordination primitives), relegating a discussion of other primitives (e.g., monitors) to the exercises. Our goal is to remove all the mystery about how primitives can be implemented on conventional hardware. Once the essential magic of a particular set of primitives is understood, the implementation of alternative versions will be easy to master.

The Xinu code presented in the text runs on many hardware platforms. We will focus on two low-cost experimenter boards that use two popular processor architectures: a *Galileo* board that contains an Intel (x86) processor and a BeagleBone Black that contains an ARM processor. The paradigm is that a programmer uses conventional tools (editor, compiler, and linker) to create a Xinu image. The image is then loaded onto a target board, and the board boots the Xinu operating system.

The book is designed for advanced undergraduate or graduate-level courses, and for computing professionals who want to understand operating systems. Although there is nothing inherently difficult about any topic, covering most of the material in one semester demands an extremely rapid pace usually unattainable by undergraduates. Few undergraduates are adept at reading code, and fewer still understand the details of a run-time environment or machine architecture. Thus, they need to be guided through the chapters on process management and process synchronization carefully. Choosing items to omit depends largely on the background of students who take the course. If time is limited, I recommend covering Chapters 1–7 (process management), 9 (basic memory management), 12 (interrupt processing), 13 (clock management), 14 (device-independent I/O), and 19 (file systems). If students have taken a data structures course that covers memory management and list manipulation, Chapters 4 and 9 can be skipped. It is important for students to understand that most operating systems include network communication. If they will take a separate course in networking, however, they can skip Chapter 17 on network protocols. The text includes chapters on both a remote disk system (18) and a remote file system (20); one of the two can be skipped. The chapter on a remote disk system may be slightly more pertinent because it introduces the topic of disk block caching, which is central in many operating systems.

In grad courses, class time can be spent discussing motivations, principles, trade-offs, alternative sets of primitives, and alternative implementations. Students should emerge with a firm understanding of the process model and the relationship between interrupts and processes as well as the ability to understand, create, and modify system components. They should have a complete mental model of the entire system, and know how all the pieces interact. Two topics should be included in both graduate and undergraduate courses: the important metamorphosis that occurs during startup when a sequential program is transformed into a process, and the transformation in the shell when a sequence of characters on an input line become string arguments passed to a command process.

In all cases, learning improves dramatically if students have hands-on experience with the system. The low cost of the boards we have selected (they are available for less than $50 US) means each student can afford to purchase a board and the cables needed to connect it to a laptop or other development computer. Ideally, they can start to use the system in the first few days or weeks of the class before they try to understand the internal structure. Chapter 1 provides a few examples and encourages experimentation. (It is surprising how many students take operating system courses without ever writing a concurrent program or using system facilities.) Many of the exercises suggest improvements, experiments, and alternative implementations. Larger projects are also possible. Examples that have been used with various hardware include: a paging system, mechanisms to synchronize execution across computers, and the design of a virtual network. Other students have transported Xinu to various processors or built device drivers for various I/O devices. Of course, a background in programming is assumed — working on the code requires a knowledge of the C programming language and a basic understanding of data structures, including linked lists, stacks, and queues.

At Purdue, we have a lab with an automated system providing access to the experimenter boards. A student uses cross-development tools on a conventional Linux system to create a Xinu image. The student then runs an application that uses the lab network to allocate one of the boards, load the image onto the board, connect the console line from the board to a window on the student's screen, and boot the image. Because the hardware is inexpensive, a lab can be constructed at very low cost. For details, contact the author or look on the website:

www.xinu.cs.purdue.edu

I owe much to my experiences, good and bad, with commercially available operating systems. Although Xinu differs internally from existing systems, the fundamental ideas are not new. Many basic ideas and names have been taken from Unix. However, readers should be aware that many of the function arguments and the internal structure of the two systems differ dramatically — applications written for one system will not run on the other without modification. Xinu is not Unix.

I gratefully acknowledge the help of many people who contributed ideas, hard work, and enthusiasm to the Xinu project. Over the years, many graduate students at Purdue have worked on the system, ported it, and written device drivers. The version in this book represents a complete rewrite, and many students at Purdue contributed. As we updated the code, we strove to preserve the elegance of the original design. Rajas Karandikar and Jim Lembke created drivers and the multi-step downloading system used on the Galileo. Students in my operating systems class, including Andres Bravo, Gregory Essertel, Michael Phay, Sang Rhee, and Checed Rodgers, found problems and contributed to the code. Special thanks go to my wife and partner, Christine, whose careful editing and suggestions made many improvements throughout.

Douglas Comer

About the Author

Douglas Comer, Distinguished Professor of Computer Science at Purdue University, is an internationally recognized expert on computer networking, the TCP/IP protocols, the Internet, and operating systems design. The author of numerous refereed articles and technical books, he is a pioneer in the development of curriculum and laboratories for research and education.

A prolific author, Dr. Comer's popular books have been translated into sixteen languages, and are used in industry as well as computer science, engineering, and business departments around the world. His landmark three-volume series *Internetworking With TCP/IP* revolutionized networking and network education. His textbooks and innovative laboratory manuals have shaped and continue to shape graduate and undergraduate curricula.

The accuracy and insight of Dr. Comer's books reflect his extensive background in computer systems. His research spans both hardware and software. He has created the Xinu operating system, written device drivers, and implemented network protocol software for conventional computers as well as network processors. Software that has resulted from Dr. Comer's research has been used by industry in a variety of products.

Dr. Comer has created and teaches courses on network protocols, operating systems, and computer architecture for a variety of audiences, including courses for engineers as well as academic audiences. His innovative educational laboratories allow him and his students to design and implement working prototypes of large, complex systems, and measure the performance of the resulting prototypes. He continues to teach at companies, universities, and conferences around the world. In addition, Dr. Comer consults for industry on the design of computer networks and systems.

For twenty years, Professor Comer served as editor-in-chief of the research journal *Software — Practice and Experience*. While on an extended leave from Purdue, he served as Vice President of Research at Cisco Systems. He is a Fellow of the ACM, a Fellow of the Purdue Teaching Academy, and a recipient of numerous awards, including a Usenix Lifetime Achievement award.

Additional information about Dr. Comer can be found at:

www.cs.purdue.edu/people/comer

and information about his books can be found at:

www.comerbooks.com

Chapter Contents

1.1 Operating Systems, 3
1.2 Approach Used In The Text, 5
1.3 A Hierarchical Design, 5
1.4 The Xinu Operating System, 7
1.5 What An Operating System Is Not, 8
1.6 An Operating System Viewed From The Outside, 9
1.7 Remainder Of The Text, 10
1.8 Perspective, 11
1.9 Summary, 11

1

Introduction And Overview

Our little systems have their day.

— Alfred, Lord Tennyson

1.1 Operating Systems

Hidden in every intelligent device and computer system is the software that controls processing, manages resources, and communicates with peripherals such as display screens, disks, computer networks, and printers. Collectively, the code that performs control and coordination chores has been referred to as an *executive*, a *monitor*, a *task manager*, and a *kernel*; we will use the broader term *operating system*.

Computer operating systems are among the most complex objects created by mankind: they allow multiple computational processes and users to share a processor simultaneously, protect data from unauthorized access, and keep independent input/output (I/O) devices operating correctly. The high-level services an operating system offers are all achieved by executing intricately detailed, low-level hardware instructions. Interestingly, an operating system is not an independent mechanism that controls a computer from the outside — it consists of software that is executed by the same processor that executes applications. In fact, when a processor is executing an application, the processor cannot be executing the operating system and vice versa.

Arranging mechanisms that guarantee an operating system will always regain control after an application runs complicates system design. The most impressive aspect of an operating system, however, arises from the difference in functionality between the services offered and underlying hardware: an operating system provides impressively high-level services over extremely low-level hardware. As the book proceeds, we will understand how crude the underlying hardware can be, and see how much system software is required to handle even a simple device such as the serial I/O device used

for a keyboard or mouse. The philosophy is straightforward: an operating system should provide abstractions that make programming easier rather than abstractions that reflect the underlying hardware. Thus, we conclude:

> *An operating system is designed to hide low-level hardware details and to create an abstract machine that provides applications with high-level services.*

Operating system design is not a well-known craft. In the beginning, because computers were scarce and expensive, only a few programmers had an opportunity to work on operating systems. By the time advances in micro-electronic technology reduced fabrication costs and made personal computers available, operating systems had become commodities, and few programmers need to work on them. Interestingly, microprocessors have become so inexpensive that most electronic devices are now constructed from programmable processors rather than from discrete logic. As a result, designing and implementing software systems for microprocessors and microcontrollers is no longer a task reserved for a few specialists; it has become a skill expected of competent systems programmers.

Fortunately, our understanding of operating systems has grown along with the technology we use to produce new machines. Researchers have explored fundamental issues, formulated design principles, identified essential components, and devised ways that components can work together. More important, researchers have identified abstractions, such as files and current processes, that are common to all operating systems, and have found efficient implementations for the abstractions. Finally, we have learned how to organize the components of an operating system into a meaningful structure that simplifies system design and implementation.

Compared to its early counterparts, a modern system is simple, clean, and portable. A well-designed system follows a basic pattern that partitions software into a set of basic components. As a result, a modern system can be easier to understand and modify, can contain less code, and has less processing overhead than early systems.

Vendors that sell large commercial operating systems include many extra software components along with an operating system. For example, a typical software distribution includes compilers, linkers, loaders, library functions, and a set of applications. To distinguish between the extras and the basic system, we sometimes use the term *kernel* to refer to the code that remains resident in memory and provides key services such as support for concurrent processes. Throughout the text, we will assume the term *operating system* refers to the kernel, and does not include all additional facilities. A design that minimizes the facilities in a kernel is sometimes called a *microkernel* design; our discussions will concentrate on a microkernel.

1.2 Approach Used In The Text

This book is a guide to the structure, design, and implementation of operating system kernels. Instead of merely surveying extant systems, listing their features, and describing functionality abstractly, the book takes an engineering approach. It explains how to construct each OS abstraction, and shows how the individual abstractions can be organized into an elegant, efficient design.

Our approach provides two advantages. First, because the text covers every part of the system, a reader will see how an entire system fits together, not merely how one or two parts interact. Second, because source code is available for all pieces described in the text, no mystery remains about any part of the implementation — a reader can obtain a copy of the system to examine, modify, instrument, measure, extend, or transport to another architecture. By the end of the book, a reader will see how each piece of an operating system fits into the design, and will be prepared to understand alternative design choices.

Our focus on implementation means that the software forms an integral part of the text. In fact, the code provides a centerpiece for discussion; one must read and study the program listings to appreciate the underlying subtlety and engineering detail. The example code is minimal, which means a reader can concentrate on concepts without wading through many pages of code. Some of the exercises suggest improvements or modifications that require a reader to delve into details or invent alternatives; a skillful programmer will find additional ways to improve and extend the system.

1.3 A Hierarchical Design

If designed well, the interior of an operating system can be as elegant and clean as the best application program. The design described in this book achieves elegance by partitioning system functions into eight major categories, and organizing the components into a multi-level hierarchy. Each level of the system provides an abstract service, implemented in terms of the abstract services provided by lower levels. The approach offers a property that will become apparent: successively larger subsets of the levels can be selected to form successively more powerful systems. We will see how a hierarchical approach provides a model for designers that helps reduce complexity.

Another important property of our approach arises from runtime efficiency — a designer can structure pieces of an operating system into a hierarchy without introducing extra overhead. In particular, our approach differs from a conventional layered system in which a function at level K can only invoke functions at level $K-1$. In our multi-level approach, the hierarchy only provides a conceptual model for a designer — at runtime, a function at a given level of the hierarchy can invoke any of the functions in lower levels directly. We will see that direct invocation makes the entire system efficient.

Figure 1.1 illustrates the hierarchy used in the text, gives a preview of the components we will discuss, and shows the structure into which all pieces are organized.

Figure 1.1 The multi-level organization used in the text.

At the heart of the hierarchy lies the computer hardware. Although not part of the operating system itself, modern hardware includes features that allow tight integration with an operating system. Thus, we think of the hardware as forming level zero of our hierarchy.

Building out from the hardware, each higher level of operating system software provides more powerful primitives that shield applications from the raw hardware. A memory manager controls and allocates memory. Process management forms the most fundamental component of the operating system, and includes a scheduler and context switch. Functions in the next level constitute the rest of the process manager, providing primitives to create, kill, suspend, and resume processes. Just beyond the process manager comes a process coordination component that implements semaphores. Functions for real-time clock management occupy the next level, and allow application software to delay for a specified time. On top of the real-time clock level lies a level of device-independent I/O routines that provide familiar services, such as *read* and *write*. Above the device routines, a level implements network communication, and the level above that implements a file system. Application programs occupy the highest conceptual level of the hierarchy — an application has access to all the facilities provided by lower levels.

The internal organization of a system should not be confused with the services the system provides. Although components are organized into levels to make the design and implementation cleaner, the resulting hierarchical structure does not restrict system calls at runtime. That is, once the system has been built, facilities from all levels of the hierarchy can be exposed to applications. For example, an application can invoke semaphore functions, such as *wait* and *signal*, that reside in the process coordination level just as easily as it can invoke functions such as *putc* that reside in an outer level. Thus, the multi-level structure describes only the internal implementation, and does not restrict the services the system provides.

1.4 The Xinu Operating System

Examples in the book are taken from the *Xinu*† operating system. Xinu is a small, elegant system that is intended for use in an embedded environment, such as a cell phone or an MP3 player. Typically, Xinu is loaded into memory along with a fixed set of applications when the system boots. Of course, if memory is constrained or the hardware architecture uses a separate memory for instructions, Xinu can be executed from Flash or other read-only memory. In a typical system, however, executing from main memory produces higher performance.

Xinu is not a toy; it is a powerful operating system that has been used in commercial products. For example, Xinu was used in pinball games sold under the Williams/Bally brand (the major manufacturer), Woodward Corporation uses Xinu to control large gas/steam and diesel/steam turbine engines, and Lexmark Corporation used Xinu as the operating system in its printers until 2005. In each case, when the device was powered on, the hardware loaded a memory image that contained Xinu.

---

†The name stands for Xinu Is Not Unix. As we will see, the internal structure of Xinu differs completely from the internal structure of Unix (or Linux). Xinu is smaller, more elegant, and easier to understand.

Xinu contains the fundamental components of an operating system, including: process, memory, and timer management mechanisms, interprocess communication facilities, device-independent I/O functions, and Internet protocol software. Xinu can control I/O devices and perform chores such as reading keystrokes from a keyboard or keypad, displaying characters on an output device, managing multiple, simultaneous computations, controlling timers, passing messages between computations, and allowing applications to access the Internet.

Xinu illustrates how the hierarchical design that is described above applies in practice. It also shows how all the pieces of an operating system function as a uniform, integrated whole, and how an operating system makes services available to application programs.

1.5 What An Operating System Is Not

Before proceeding into the design of an operating system, we should agree on what we are about to study. Surprisingly, many programmers do not have a correct intuitive definition of an operating system. Perhaps the problem arises because vendors and computer professionals often apply the terminology broadly to refer to all software supplied by a vendor as well as the operating system itself, or perhaps confusion arises because few programmers access system services directly. In any case, we can clarify the definition quickly by ruling out well-known items that are not part of the operating system kernel.

First, an operating system is not a language or a compiler. Of course, an operating system must be written in some language, and languages have been designed that incorporate operating systems features and facilities. Further confusion arises because a software vendor may offer one or more compilers that have been integrated with their operating system. However, an operating system does not depend on an integrated language facility — we will see that a system can be constructed using a conventional language and a conventional compiler.

Second, an operating system is not a windowing system or a browser. Many computers and electronic devices have a screen that is capable of displaying graphics, and sophisticated systems permit applications to create and control multiple, independent windows. Although windowing mechanisms rely on an operating system, a windowing system can be replaced without replacing the operating system.

Third, an operating system is not a command interpreter. Embedded systems often include a *Command Line Interface* (*CLI*); some embedded systems rely on a CLI for all control. In a modern operating system, however, the command interpreter operates as an application program, and the interpreter can be changed without modifying the underlying system.

Fourth, an operating system is not a library of functions or methods. Almost all application programs use library functions, and the software found in libraries can offer significant convenience and functionality. Some operating systems even employ an optimization that allows code from a library to be loaded in memory and shared among all

applications. Despite the close relationship, library software remains independent of the underlying operating system.

Fifth, an operating system is not the first code that runs after a computer is powered on. Instead, the computer contains *firmware* (i.e., a program in non-volatile memory) that initializes various pieces of hardware, loads a copy of the operating system into memory, and then jumps to the beginning of the operating system. On a PC, for example, the firmware is known as the *Basic Input Output System* (*BIOS*). We will learn more about bootstrapping in Chapter 22.

1.6 An Operating System Viewed From The Outside

The essence of an operating system lies in the services it provides to applications. An application accesses operating system services by making *system calls*. In source code, a system call appears to be a conventional function invocation. At runtime, however, a system call and a conventional function call differ. Instead of transferring control to another function, a system call transfers control to the operating system, which performs the requested service for the application. Taken as a set, system calls establish a well-defined boundary between applications and the underlying operating system that is known as an *Application Program Interface* (*API*)†. The API defines the services that the system provides as well as the details of how an application uses the services.

To appreciate the interior of an operating system, one must first understand the characteristics of the API and see how applications use the services. This chapter introduces a few fundamental services, using examples from the Xinu operating system to illustrate the concepts. For example, the Xinu function *putc* writes a single character to a specified I/O device. *Putc* takes two arguments: a device identifier and a character to write. File *ex1.c* contains an example C program that writes the message "hi" on the console when run under Xinu:

```
/* ex1.c - main */

#include <xinu.h>

/*------------------------------------------------------------------------
 * main  -  Write "hi" on the console
 *------------------------------------------------------------------------
 */
void    main(void)
{
        putc(CONSOLE, 'h');
        putc(CONSOLE, 'i');
        putc(CONSOLE, '\n');
}
```

†The interface is also known as a *system call interface* or the *kernel interface*.

The code introduces several conventions used throughout Xinu. The statement:

```
#include <xinu.h>
```

inserts a set of declarations into a source program that allows the program to reference operating system parameters. For example, the Xinu configuration file defines symbolic constant *CONSOLE* to correspond to a console serial device a programmer uses to interact with the embedded system. Later, we will see that *xinu.h* contains a series of *#include* statements that reference files needed by the Xinu system, and we will learn how names like *CONSOLE* become synonymous with devices; for now, it is sufficient to know that the *include* statement must appear in any Xinu application.

To permit communication with an embedded system (e.g., for debugging), the serial device on the embedded system can be connected to a *terminal application* on a conventional computer. Each time a user presses a key on the computer's keyboard, the terminal application sends the keystroke over the serial line to the embedded system. Similarly, each time the embedded system sends a character to the serial device, the terminal application displays the character on the user's screen. Thus, a console provides two-way communication between the embedded system and the outside world.

The main program listed above writes three characters to the console serial device: "h", "i", and a line feed (known as *NEWLINE*). The line feed is a control character that moves the cursor to the beginning of the next line. Xinu does not perform any special operations when the program sends control characters — control characters are merely passed on to the serial device just like alphanumeric characters. A control character has been included in the example to illustrate that *putc* is not line-oriented; in Xinu, a programmer is responsible for terminating a line.

The example source file introduces two important conventions followed throughout the book. First, the file begins with a one-line comment that contains the name of the file (*ex1.c*). If a source file contains multiple functions, the name of each appears on the comment line. Knowing the names of files will help you locate them in a machine-readable copy of Xinu. Second, the file contains a block comment that identifies the start of each function (*main*). Having a block comment before each function makes it easy to locate functions in a given file.

1.7 Remainder Of The Text

The remainder of the text proceeds through the design of a system that follows the multi-level organization that Figure 1.1 illustrates. Chapter 2 describes concurrent programming and the services an operating system supplies. Successive chapters consider the levels in roughly the same order as they are designed and built: from the innermost outward. Each chapter explains the role of one level in the system, describes new abstractions, and illustrates the details with source code. Taken together, the chapters describe a complete, working system and explain how the components fit together in a clean and elegant design.

Although the bottom-up approach may seem awkward at first, it shows how an operating system designer builds a system. The overall structure of the system will start to become clear by Chapter 9. By the end of Chapter 14, readers will understand a minimal kernel capable of supporting concurrent programs. By Chapter 20, the system will include remote file access, and by Chapter 22, the design will include a complete set of operating system functions and code to initialize the entire system.

1.8 Perspective

Why study operating systems? It may seem pointless because commercial systems are widely available and relatively few programmers write operating system code. However, a strong motivation exists: even in small embedded systems, applications run on top of an operating system and use the services it provides. Therefore, understanding how an operating system works internally helps a programmer appreciate concurrent processing and make sensible choices about system services. In some cases, the behavior of application software can only be understood, and problems can only be solved, by understanding how an operating system manages concurrent process execution.

The key to learning operating systems lies in persistence. A concurrent paradigm requires you to think about computer programs in new ways. Until you grasp the basics, it may seem confusing. Fortunately, you will not be overwhelmed with code — some of the most important ideas are contained neatly in a few lines of code. Once you understand what's going on, you will be able to read operating systems code easily, understand why process coordination is needed, and see how functions work together. By the end of the text, you will be able to write or modify operating systems functions.

1.9 Summary

An operating system provides a set of convenient, high-level services over low-level hardware. Because most applications use operating system services, programmers need to understand operating system principles. Programmers who work on embedded devices need to understand operating system design more deeply. Using a hierarchical structure can make an operating system easier to design, understand, and modify.

The text takes a practical approach. Instead of merely describing commercial systems or listing operating system features, it uses an example system, Xinu, to illustrate how a system can be designed. Although it is small and elegant, Xinu is not a toy — it has been used in commercial products. Xinu follows a multi-level design in which software components are organized into eight conceptual levels. The text explains one level of the system at a time, beginning with the raw hardware and ending with a working operating system.

EXERCISES

1.1 Should an operating system make hardware facilities available to application programs? Why or why not?

1.2 What are the advantages of using a real operating system in examples?

1.3 What are the eight major components of an operating system?

1.4 In the Xinu multi-level hierarchy, can a function in the file system code use a function in the process manager? Can a function in the process manager use a function in the file system? Explain.

1.5 Explore the system calls available on your favorite operating system, and write a program that uses them.

1.6 Various programming languages have been designed that incorporate OS concepts such as processes and process synchronization primitives. Find an example language, and make a list of the facilities it offers.

1.7 Search the web, and make a list of the major commercial operating systems that are in use.

1.8 Compare the facilities in Linux and Microsoft's Windows operating systems. Does either one support functionality that is not available in the other?

1.9 The set of functions that an operating system makes available to application programs is known as the *Application Program Interface* or the *system call interface*. Choose two example operating systems, count the functions in the interface that each makes available, and compare the counts.

1.10 Extend the previous exercise by identifying functions that are available in one system but not in the other. Characterize the purpose and importance of the functions.

1.11 How large is an operating system? Choose an example system, and count the lines of source code used for the kernel.

Chapter Contents

2.1 Introduction, 15
2.2 Programming Models For Multiple Activities, 16
2.3 Operating System Services, 17
2.4 Concurrent Processing Concepts And Terminology, 17
2.5 Distinction Between Sequential And Concurrent Programs, 19
2.6 Multiple Processes Sharing A Single Piece Of Code, 21
2.7 Process Exit And Process Termination, 23
2.8 Shared Memory, Race Conditions, And Synchronization, 24
2.9 Semaphores And Mutual Exclusion, 28
2.10 Type Names Used In Xinu, 30
2.11 Operating System Debugging With Kputc And Kprintf, 31
2.12 Perspective, 32
2.13 Summary, 32

2

Concurrent Execution And Operating System Services

From an article on a new operating system for the IBM PC: Real concurrency — in which one program actually continues to function while you call up and use another — is more amazing but of small use to the average person. How many programs do you have that take more than a few seconds to perform any task?

— New York Times, 25 April 1989

2.1 Introduction

This chapter considers the concurrent programming environment that an operating system provides for applications. It describes a model of concurrent execution, and shows why applications that operate concurrently need mechanisms to coordinate and synchronize. It introduces basic concepts, such as processes and semaphores, and explains how applications use such facilities.

Instead of describing operating systems abstractly, the chapter uses concrete examples from the Xinu system to illustrate concepts such as concurrency and synchronization. The chapter contains trivial applications that capture the essence of concurrent execution in a few lines of code. Later chapters expand the discussion by explaining in detail how an operating system implements each of the facilities described.

2.2 Programming Models For Multiple Activities

Even small computing devices are designed to handle multiple tasks at the same time. For example, while a voice call is connected, a cell phone can display the time of day, accept text messages, and allow the user to adjust the volume. More complex computing systems allow a user to run multiple applications that execute at the same time. The question arises: how should the software in such systems be organized? Three basic approaches can be used:

- Synchronous event loop
- Asynchronous event handlers
- Concurrent execution

Synchronous event loop. The term *synchronous* refers to events that are coordinated. A *synchronous event loop* uses a single, large iteration to handle coordination. During a given iteration of the loop, the code checks each possible activity and invokes the appropriate handler. Thus, the code has a structure similar to the following:

```
while (1) {   /* synchronous loop runs forever */
        Update time-of-day clock;
        if (screen timeout has expired) {
                turn off the screen;
        }
        if (volume button is being pushed) {
                adjust volume;
        }
        if (text message has arrived) {
                Display notification for user;
        }
        ...
}
```

Asynchronous event handlers. An asynchronous paradigm is used in systems where the hardware can be configured to invoke a *handler* for each event. For example, the code to adjust volume might be placed in memory at location 100, and the hardware is configured so that when the *volume* button is pressed, control transfers to location 100. Similarly, the hardware can be configured so that when a text message arrives, control transfers to location 200, and so on. A programmer writes a separate piece of code for each event, and uses global variables to coordinate the interactions. For example, if a user presses the *mute* button, the code associated with the mute event turns off the audio and records the status in a global variable. Later, when the user adjusts the volume, code associated with the volume button checks the global variable, turns on the audio, and changes the global variable to indicate that audio is on.

Concurrent execution. The third paradigm used to organize multiple activities is the most significant: software is organized as a set of programs that each operate concurrently. The model is sometimes called *run-to-completion* because each computation appears to run until it chooses to stop. From a programmer's point of view, concurrent execution is a delight. Compared to synchronous or asynchronous events, concurrent execution is more powerful, easier to understand, and less error-prone.

The next sections describe operating systems that provide the support needed for concurrency, and characterize the concurrent model. Later chapters examine the underlying operating system mechanisms and functions that enable a concurrent programming model.

2.3 Operating System Services

What are the main services that an operating system supplies? Although the details vary from system to system, most systems supply the same basic services. The services (with the chapters of the text that describe them) are:

- Support for concurrent execution (5–6)
- Facilities for process synchronization (7)
- Inter-process communication mechanisms (8 and 11)
- Dynamic memory allocation (9)
- Management of address spaces and virtual memory (10)
- High-level interface for I/O devices (13–15)
- Network and Internet communication (16–17)
- A file system and file access facilities (19–21)

Concurrent execution is at the heart of an operating system, and we will see that concurrency affects each piece of operating system code. Thus, we begin by examining the facilities an operating system offers for concurrency, and use concurrency to show how an application program invokes services.

2.4 Concurrent Processing Concepts And Terminology

Conventional programs are called *sequential* because a programmer imagines a computer executing the code statement by statement; at any instant, the machine is executing exactly one statement. Operating systems support an extended view of computation called *concurrent processing*. Concurrent processing means that multiple computations can proceed "at the same time."

Many questions arise about concurrent processing. It is easy to imagine N independent programs being executed simultaneously by N processors (i.e., N cores), but it is difficult to imagine a set of independent computations proceeding simultaneously

on a computer that has fewer than N processing units. Is concurrent computation possible even if a computer has a single core? If multiple computations each proceed simultaneously, how does the system keep one program from interfering with others? How do the programs cooperate so that only one takes control of an input or output device at a given time?

Although many processors do incorporate some amount of parallelism, the most visible form of concurrency, multiple independent applications that execute simultaneously, is a grand illusion. To create the illusion, an operating system uses a technique, called *multitasking* or *multiprogramming* — the operating system switches the available processor(s) among multiple programs, allowing a processor to execute one program for only a few milliseconds before moving on to another. When viewed by a human, the programs all appear to proceed. Multitasking forms the basis of most operating systems. The only exceptions are systems used in basic embedded devices, such as a simplistic remote control used with a television, and safety-critical systems, such as flight avionics and medical device controllers. In such cases, designers use a synchronous event loop rather than a multitasking system, either to reduce cost or to guarantee that tight time constraints can be met absolutely.

Systems that support multitasking can be divided into two broad categories:

- Timesharing
- Real-time

Timesharing. A *timesharing system* gives equal priority to all computations, and permits computations to start or terminate at any time. Because they allow computations to be created dynamically, timesharing systems are popular for computers that human users operate. A timesharing system allows a human to leave an email application running and a background application playing music while using a browser to view a web page. The chief characteristic of a timesharing system is that the amount of processing a computation receives is inversely proportional to the load on the system — if N computations are executing, each computation receives approximately $1/N$ of the available processor cycles. Thus, as more computations appear, each proceeds at a slower rate.

Real-time. Because it is designed to meet performance constraints, a *real-time system* does not treat all computations equally. Instead, a real-time system assigns priorities to computations, and schedules the processor carefully to ensure that each computation meets its required schedule. The chief characteristic of a real-time system arises from its ability to give the processor to high-priority tasks, even if other tasks are waiting. For example, by giving priority to voice transmission, a real-time system in a cell phone can guarantee that the conversation is uninterrupted, even if a user runs an application to view the weather or an application to play a game.

Designers of multitasking systems have used a variety of terms to describe a single computation, including *process*, *task*, *job*, and *thread of control*. The terms *process* and *job* often connote a single computation that is self-contained and isolated from other computations. Typically, a process occupies a separate region of memory, and the operating system prevents a process from accessing the memory that has been assigned

to another process. The term *task* refers to a process that is declared statically. That is, a programming language allows a programmer to declare a task similar to the way one declares a function. The term *thread* refers to a type of concurrent process that shares an address space with other threads. Shared memory means that members of the set can exchange information efficiently. Early scientific literature used the term *process* to refer to concurrent execution in a generic sense; the Unix operating system popularized the idea that each process occupied a separate address space. The Mach system introduced a two-level concurrent programming scheme in which the operating system allows a user to create one or more processes that each operate in an independent region of memory, and to create multiple threads of control within each process. Linux follows the Mach model. To refer to a Linux-style process, the word *Process* is written with an uppercase *P*.

Because it is designed for an embedded environment, Xinu permits processes to share an address space. To be precise, we might say that Xinu processes follow a *thread* model. However, because the term *process* is widely accepted, we will use it throughout the text to refer generically to a concurrent computation.

The next section helps distinguish concurrent execution from sequential execution by examining a few applications. As we will see, the difference plays a central role in operating system design — each piece of an operating system must be built to support concurrent execution.

2.5 Distinction Between Sequential And Concurrent Programs

When a programmer creates a conventional (i.e., sequential) program, the programmer imagines a single processor executing the program step-by-step without interruption or interference. When writing code that will be executed concurrently, however, a programmer must take a different view and imagine multiple computations executing simultaneously. The code inside an operating system provides an excellent example of code that must accommodate concurrency. At any given instant, multiple processes may be executing. In the simplest case, each process executes application code that no other process is executing. However, an operating system designer must plan for a situation in which multiple processes have invoked a single operating system function, or even a case where multiple processes are executing the same instruction. To further complicate matters, the operating system may switch the processor among processes at any time; in a multitasking system, no guarantee can be made about the relative speed at which a given computation will proceed.

Designing code to operate correctly in a concurrent environment provides a tough intellectual challenge because a programmer must ensure that all processes perform the intended function correctly, no matter what operating system code they execute or in which order they execute. We will see how the notion of concurrent execution affects each line of code in an operating system.

To understand applications in a concurrent environment, consider the Xinu model. When it boots, Xinu creates a single concurrent process that starts running the main pro-

gram. The initial process can continue execution by itself, or it can create additional processes. When a new process is created, the original process continues to execute, and the new process executes concurrently. Either the original process or a new process can create additional processes that execute concurrently.

As an example, consider the code for a concurrent main program that creates two additional processes. Each of the two new processes sends characters over the console serial device: the first process sends the letter *A*, and the second sends the letter *B*. File *ex2.c* contains the source code, which consists of a main program and two functions, *sndA* and *sndB*.

```
/* ex2.c - main, sndA, sndB */

#include <xinu.h>

void    sndA(void), sndB(void);

/*------------------------------------------------------------------------
 * main  -  Example of creating processes in Xinu
 *------------------------------------------------------------------------
 */
void    main(void)
{
        resume( create(sndA, 1024, 20, "process 1", 0) );
        resume( create(sndB, 1024, 20, "process 2", 0) );
}

/*------------------------------------------------------------------------
 * sndA  -  Repeatedly emit 'A' on the console without terminating
 *------------------------------------------------------------------------
 */
void    sndA(void)
{
        while( 1 )
                putc(CONSOLE, 'A');
}

/*------------------------------------------------------------------------
 * sndB  -  Repeatedly emit 'B' on the console without terminating
 *------------------------------------------------------------------------
 */
void    sndB(void)
{
        while( 1 )
                putc(CONSOLE, 'B');
}
```

In the code, the main program never calls either function directly. Instead, the main program calls two operating system functions, *create* and *resume*. Each call to *create* forms a new process that will begin executing instructions at the address specified by its first argument. In the example, the first call to *create* passes the address of function *sndA*, and the second call passes the address of function *sndB*.† Thus, the code creates a process to execute *sndA* and a process to execute *sndB*. *Create* establishes a process, leaves the process ready to execute but temporarily suspended, and returns an integer value that is known as a *process identifier* or *process ID*. The operating system uses the process ID to identify the newly created process; an application uses the process ID to reference the process. In the example, the main program passes the ID returned by *create* to *resume* as an argument. *Resume* starts (i.e., unsuspends) the process, allowing the process to begin execution. The distinction between normal function calls and process creation is:

> *A normal function call does not return until the called function completes. Process creation functions* create *and* resume *return to the caller immediately after starting a new process, which allows execution of both the existing process and the new process to proceed concurrently.*

In Xinu, all processes execute concurrently. That is, execution of a given process continues independent of other processes unless a programmer explicitly controls interactions among processes. In the example, the first new process executes code in function *sndA*, sending the letter *A* continuously, and the second executes code in function *sndB*, sending the letter *B* continuously. Because the processes execute concurrently, the output is a mixture of *A*s and *B*s.

What happens to the main program? Remember that in an operating system, each computation corresponds to a process. Therefore, we should ask, "What happens to the process executing the main program?" Because it has reached the end of the main program, the process executing the main program exits after the second call to *resume*. Its exit does not affect the newly created processes — they continue to send *A*s and *B*s. A later section describes process termination in more detail.

2.6 Multiple Processes Sharing A Single Piece Of Code

The example in file *ex2.c* shows each process executing a separate function. It is possible, however, for multiple processes to execute the same function. Arranging for processes to share code can be essential in an embedded system that has a small memory. To see an example of processes sharing code, consider the program in file *ex3.c*.

---

†Other arguments to *create* specify the stack space needed, a scheduling priority, a name for the process, the count of arguments passed to the process, and (when applicable) the argument values passed to the process; we will see details later.

```
/* ex3.c - main, sndch */

#include <xinu.h>

void    sndch(char);

/*------------------------------------------------------------------------
 * main  -  Example of 2 processes executing the same code concurrently
 *------------------------------------------------------------------------
 */
void    main(void)
{
        resume( create(sndch, 1024, 20, "send A", 1, 'A') );
        resume( create(sndch, 1024, 20, "send B", 1, 'B') );
}

/*------------------------------------------------------------------------
 * sndch  -  Output a character on a serial device indefinitely
 *------------------------------------------------------------------------
 */
void    sndch(
          char  ch              /* The character to emit continuously   */
        )
{
        while ( 1 )
                putc(CONSOLE, ch);
}
```

As in the previous example, a single process begins executing the main program. The process calls *create* twice to start two new processes that each execute code from function *sndch*. The final two arguments in the call to *create* specify that *create* will pass one argument to the newly created process and a value for the argument. Thus, the first process receives the character *A* as an argument, and the second process receives character *B*.

Although they execute the same code, the two processes proceed concurrently without any effect on one another. In particular, each process has its own copy of arguments and local variables. Thus, one process emits *A*s, while the other process emits *B*s. The key point is:

> *A program consists of code executed by a single process of control. In contrast, concurrent processes are not uniquely associated with a piece of code; multiple processes can execute the same code simultaneously.*

The examples provide a hint of the difficulty involved in designing an operating system. Not only must each piece be designed to operate correctly by itself, the designer must also guarantee that multiple processes can execute a given piece of code concurrently without interfering with one another.

Although processes can share code and global variables, each process must have a private copy of local variables. To understand why, consider the chaos that would result if all processes shared every variable. If two processes tried to use a shared variable as the index of a *for* loop, for example, one process might change the value while another process was in the midst of executing the loop. To avoid such interference, the operating system creates an independent set of local variables for each process.

Function *create* also allocates an independent set of arguments for each process, as the example in file *ex3.c* demonstrates. In the call to *create*, the last two arguments specify a count of values that follow (1 in the example), and the value that the operating system passes to the newly created process. In the code, the first new process has character *A* as an argument, and the process begins execution with formal parameter *ch* set to *A*. The second new process begins with *ch* set to *B*. Thus, the output contains a mixture of both letters. The example points out a significant difference between the sequential and concurrent programming models.

Storage for local variables, function arguments, and a function call stack is associated with the process executing a function, not with the code for the function.

The important point is: an operating system must allocate additional storage for each process, even if the process shares the same code that another process or processes are using. As a consequence, the amount of memory available limits the number of processes that can be created.

2.7 Process Exit And Process Termination

The example in file *ex3.c* consists of a concurrent program with three processes: the initial process and the two processes that were started with the system call *create*. Recall that when it reaches the end of the code in the main program, the initial process ceases execution. We use the term *process exit* to describe the situation. Each process begins execution at the start of a function. A process can exit by reaching the end of the function or by executing a return statement in the function in which it starts. Once a process exits, it disappears from the system; there is simply one less computation in progress.

Process exit should not be confused with normal function call and return or with recursive function calls. Like a sequential program, each process has its own stack of function calls. Whenever it executes a call, an activation record for the called function is pushed onto the stack. Whenever it returns, a function's activation record is popped

off the stack. Process exit occurs only when the process pops the last activation record (the one that corresponds to the top-level function in which the process started) off its stack.

The system routine *kill* provides a mechanism to *terminate* a process without waiting for the process to exit. In a sense, *kill* is the inverse of *create* — *kill* takes a process ID as an argument, and removes the specified process immediately. A process can be killed at any time and at any level of function nesting. When terminated, the process ceases execution and local variables that have been allocated to the process disappear; in fact, the entire stack of functions for the process is removed.

A process can exit by killing itself as easily as it can kill another process. To do so, the process uses system call *getpid* to obtain its own process ID, and then uses *kill* to request termination:

```
kill( getpid() );
```

When used to terminate the current process, the call to *kill* never returns because the calling process exits.

2.8 Shared Memory, Race Conditions, And Synchronization

In Xinu, each process has its own copy of local variables, function arguments, and function calls, but all processes share the set of global (external) variables. Sharing data is sometimes convenient, but it can be dangerous, especially for programmers who are unaccustomed to writing concurrent programs. For example, consider two concurrent processes that each increment a shared integer, *n*. In terms of the underlying hardware, incrementing an integer requires three steps:

- Load the value from variable *n* in memory into a register
- Increment the value in the register
- Store the value from the register back into the memory location for *n*

Because the operating system can choose to switch from one process to another at any time, a potential *race condition* exists in which two processes attempt to increment *n* at the same time. Process 1 might start first and load the value of *n* into a register. But just at that moment, the operating system switches to process 2, which loads *n*, increments the register, and stores the result. Unfortunately, when the operating system switches back to process 1, execution resumes with the original value of *n* in a register. Process 1 increments the original value of *n* and stores the result to memory, overwriting the value that process 2 placed in memory.

To see how sharing works, consider the code in file *ex4.c*. The file contains code for two processes that communicate through a shared integer, *n*†. One process repeatedly increments the value of the shared integer, while the other process repeatedly prints the value.

---

†The code uses the type name *int32* to emphasize that variable *n* is a 32-bit integer; a later section explains conventions for type names.

```
/* ex4.c - main, produce, consume */

#include <xinu.h>

void    produce(void), consume(void);

int32   n = 0;        /* Global variables are shared by all processes   */

/*------------------------------------------------------------------------
 *  main  -  Example of unsynchronized producer and consumer processes
 *------------------------------------------------------------------------
 */
void    main(void)
{
        resume( create(consume, 1024, 20, "cons", 0) );
        resume( create(produce, 1024, 20, "prod", 0) );
}

/*------------------------------------------------------------------------
 * produce  -  Increment n 2000 times and exit
 *------------------------------------------------------------------------
 */
void    produce(void)
{
        int32   i;

        for( i=1 ; i<=2000 ; i++ )
                n++;
}

/*------------------------------------------------------------------------
 * consume  -  Print n 2000 times and exit
 *------------------------------------------------------------------------
 */
void    consume(void)
{
        int32   i;

        for( i=1 ; i<=2000 ; i++ )
                printf("The value of n is %d \n", n);
}
```

In the code, global variable *n* is a shared integer that is initialized to zero. The process executing *produce* iterates 2000 times, incrementing *n*; we call the process the

producer. The process executing *consume* also iterates 2000 times. It displays the value of *n* in decimal; we call the process executing *consume* the *consumer*.

Try running file *ex4.c* — its output may surprise you. Most programmers suspect that the consumer will print at least a few, perhaps all, of the values between 0 and 2000, but it does not. In a typical run, *n* has the value 0 for the first few lines; after that, its value becomes 2000.† Even though the two processes run concurrently, they do not require the same amount of processor time per iteration. The consumer process must format and write a line of output, an operation that requires hundreds of machine instructions. Although formatting is expensive, it does not dominate the timing; output does. The consumer quickly fills the available output buffers, and must wait for the output device to send characters to the console before it can proceed. While the consumer waits, the producer runs. Because it executes only a few machine instructions per iteration, the producer runs through its entire loop and exits in the short time it takes the console device to send a line of characters. When the consumer resumes execution again, it finds that *n* has the value 2000.

Production and consumption of data by independent processes is common. The question arises: how can a programmer synchronize producer and consumer processes so the consumer receives every data value produced? Clearly, the producer must wait for the consumer to access the data item before generating another. Likewise, the consumer must wait for the producer to manufacture the next item. For the two processes to coordinate correctly, a synchronization mechanism must be designed carefully. The crucial constraint is:

> *In a concurrent programming system, no process should use the processor while waiting for another process.*

A process that executes instructions while waiting for another is said to engage in *busy waiting*. To understand our prohibition on busy waiting, think of the implementation. If a process uses the processor while waiting, the processor cannot be executing other processes. At best, the computation will be delayed unnecessarily, and at worst, the waiting process will use all the available processor cycles in a single-core system, and will wind up waiting forever.

Many operating systems include coordination functions that applications can use to avoid busy waiting. Xinu provides a *semaphore* abstraction — the system supplies a set of system calls that allow applications to operate on semaphores and to create semaphores dynamically. A semaphore consists of an integer value that is initialized when the semaphore is created and a set of zero or more processes that are waiting on the semaphore. The system call *wait* decrements a semaphore and adds the calling process to the set of waiting processes if the result is negative. The system call *signal* performs the opposite action by incrementing the semaphore and allowing one of the waiting process to continue, if any are waiting. To synchronize, a producer and consumer need two semaphores: one on which the consumer waits and one on which the producer waits. In Xinu, semaphores are created dynamically with the system call *semcreate*,

†The example assumes a 32-bit architecture in which each operation affects the entire 32-bit integer; when run on an 8-bit architecture, some bytes of *n* may be updated before others.

which takes the desired initial count as an argument, and returns an integer identifier by which the semaphore is known.

Consider the example in file *ex5.c*†. The main process creates two semaphores, *consumed* and *produced*, and passes them as arguments to the processes it creates. Because the semaphore named *produced* begins with a count of 1, *wait* will not block the first time it is called in *cons2*. So, the consumer is free to print the initial value of *n*. However, semaphore *consumed* begins with a count of 0, so the first call to *wait* in *prod2* blocks. In effect, the producer waits for semaphore *consumed* before incrementing *n* to guarantee that the consumer has printed it. When the example executes, the producer and consumer coordinate, and the consumer prints all values of *n* from 0 through 1999.

```
/* ex5.c - main, prod2, cons2 */

#include <xinu.h>

void    prod2(sid32, sid32), cons2(sid32, sid32);

int32   n = 0;                  /* Variable n has initial value zero    */

/*------------------------------------------------------------------------
 *  main  -  Producer and consumer processes synchronized with semaphores
 *------------------------------------------------------------------------
 */
void    main(void)
{
        sid32   produced, consumed;

        consumed = semcreate(0);
        produced = semcreate(1);
        resume( create(cons2, 1024, 20, "cons", 2, consumed, produced) );
        resume( create(prod2, 1024, 20, "prod", 2, consumed, produced) );
}

/*------------------------------------------------------------------------
 * prod2  -  Increment n 2000 times, waiting for it to be consumed
 *------------------------------------------------------------------------
 */
void    prod2(
          sid32         consumed,
          sid32         produced
        )
{
        int32   i;
```

†Section 2.10 on page 30 explains the type *sid32* used in the code to declare a semaphore ID.

```
        for( i=1 ; i<=2000 ; i++ ) {
                wait(consumed);
                n++;
                signal(produced);
        }
}

/*------------------------------------------------------------------------
 * cons2  -  Print n 2000 times, waiting for it to be produced
 *------------------------------------------------------------------------
 */
void    cons2(
          sid32         consumed,
          sid32         produced
        )
{
        int32   i;

        for( i=1 ; i<=2000 ; i++ ) {
                wait(produced);
                printf("n is %d \n", n);
                signal(consumed);
        }
```

2.9 Semaphores And Mutual Exclusion

Semaphores provide another important purpose, *mutual exclusion*. Two or more processes engage in mutual exclusion when they cooperate so that only one of them obtains access to a shared resource at a given time. For example, suppose two executing processes each need to insert items into a shared linked list. If they access the list concurrently, pointers can be set incorrectly. Producer–consumer synchronization does not handle the problem because the two processes do not alternate accesses. Instead, a mechanism is needed that allows either process to access the list at any time, but guarantees mutual exclusion so that one process will wait until the other finishes.

To provide mutual exclusion for use of a resource such as a linked list, the processes create a single semaphore that has an initial count of 1. Before accessing the shared resource, a process calls *wait* on the semaphore, and calls *signal* after it has completed access. The calls to *wait* and *signal* can be placed at the beginning and end of the functions designed to perform the update, or they can be placed around the lines of code that access the shared resource. We use the term *critical section* to refer to the code that cannot be executed by more than one process at a time.

For example, file *ex6.c* shows a function that adds an item to an array that is shared by multiple concurrent processes. The critical section consists of the single line:

```
shared[n++] = item;
```

which adds the item to the array and increments the count of items. Thus, the code for mutual exclusion only needs to surround one line of code. In the example, the critical section has been placed in a function, *additem*, which means the calls to *wait* and *signal* occur at the beginning and end of the function.

The code in *additem* calls *wait* on semaphore *mutex* before accessing the array, and calls *signal* on the semaphore when access is complete. In addition to the function, the file contains declarations for three global variables: an array, *shared*, an index for the array, *n*, and the ID of the semaphore used for mutual exclusion, *mutex*.

```
/* ex6.c - additem */

#include <xinu.h>

sid32   mutex;                  /* Assume initialized with semcreate    */
int32   shared[100];            /* An array shared by many processes    */
int32   n = 0;                  /* Count of items in the array          */

/*------------------------------------------------------------------------
 * additem  -  Obtain exclusive use of array shared and add an item to it
 *------------------------------------------------------------------------
 */
void    additem(
          int32         item    /* Item to add to shared array          */
        )
{
        wait(mutex);
        shared[n++] = item;
        signal(mutex);
}
```

The code assumes that global variable *mutex* will be assigned the ID of a semaphore before any calls to *additem* occur. That is, during initialization, the following statement was executed:

```
mutex = semcreate(1);
```

The code in file *ex6.c* provides a final illustration of the difference between the way one programs in sequential and concurrent environments. In a sequential program, a function often acts to isolate changes to a data structure. By localizing the code that

changes a data structure in one function, a programmer gains a sense of security — only a small amount of code needs to be checked for correctness because nothing else in the program will interfere with the data structure. In a concurrent execution environment, isolating the code into a single function is insufficient. A programmer must guarantee that execution is exclusive because interference can come from another process executing the same function at the same time.

2.10 Type Names Used In Xinu

Data declarations in the code above illustrate conventions used throughout the text. For example, semaphores are declared using the type name *sid32*. This section explains the reasoning for the conventions.

Two important questions arise when programming in C. When is it appropriate to define a new type name? How should a type name be chosen? The questions require careful thought because types fill two conceptual roles.

- Size. A type defines the storage associated with a variable and the set of values that can be assigned to the variable.
- Use. A type defines the abstract meaning of a variable and helps a programmer know how a variable can be used.

Size. The sizes of variables are especially important in an embedded system because a programmer must design data structures that fit in the memory available. Furthermore, choosing a size that does not match the underlying hardware can result in unexpected processing overhead (e.g., arithmetic operations on large integers can require multiple hardware steps). Unfortunately, C does not specify the exact size of concrete types, such as *int*, *short*, and *long*. Instead, the size of items depends on the underlying computer architecture. For example, a *long* integer can occupy 32 bits on one computer and 64 bits on another computer. To guarantee sizes, a programmer can define and use a set of type names, such as *int32*, that specify data size.

Use. The classic purpose of a type arises from the need to define the purpose of a variable (i.e., to tell how the variable is used). For example, although semaphore IDs are integers, defining a type name such as *sid32* makes it clear to anyone reading the code that a variable holds a semaphore ID and should only be used where a semaphore ID is appropriate (e.g., as an argument to a function that operates on a semaphore). Thus, although it consists of an integer, a variable of type *sid32* should not be used as a temporary value when computing an arithmetic expression, nor should it be used to store a process ID or a device ID.

Include files further complicate type declarations in C. In principle, one would expect each include file to contain the type, constant, and variable declarations related to a single module. Thus, one would expect to find the type for a process identifier declared in the include file that defines items related to processes. In an operating system, however, many cross-references exist among modules. For example, we will see that the in-

clude file for semaphores references the process type, and the include file for processes references the semaphore type.

We have chosen an approach in which types accommodate the need to define size as well as the need to define use. Instead of using the C form of *char*, *short*, *int*, and *long*, the code uses the types that Figure 2.1 lists.

Type	Meaning
byte	unsigned 8-bit value
bool8	8-bit value used as a Boolean
int16	signed 16-bit integer
uint16	unsigned 16-bit integer
int32	signed 32-bit integer
uint32	unsigned 32-bit integer

Figure 2.1 Basic type names for integers used throughout Xinu.

For types that correspond to operating system abstractions, each name combines a short mnemonic that identifies the purpose plus a numeric suffix that identifies the size. Thus, a type that defines a semaphore ID to be a 32-bit integer has been assigned the name *sid32*, and a type that defines a queue ID to be a 16-bit integer has been assigned the name *qid16*.

To permit cross-references of types among modules, a single include file, *kernel.h*, contains declarations for all type names, including the types listed in Figure 2.1. Thus, each source file must include *kernel.h* before referencing any type name. In particular, an include for *kernel.h* must precede the include statements for other modules. For convenience, a single include file, *xinu.h*, includes all header files used in Xinu in the correct order.

2.11 Operating System Debugging With Kputc And Kprintf

The examples in this chapter use Xinu functions *putc* and *printf* to display output on the *CONSOLE*. Although such functions work well once an operating system has been completed and tested, they are not used during construction or debugging because they require many components of the operating system to function correctly. What do operating system designers use?

Designers use *polled I/O*. That is, a designer creates a special I/O function that does not need interrupts to be working. Following Unix tradition, we call the special function *kputc* (i.e., a version of *putc* suitable for use inside the operating system kernel). *Kputc* takes a character, *c*, as an argument and performs four steps:

- Disable interrupts
- Wait for the CONSOLE serial device to be idle
- Send character *c* to the serial device
- Restore interrupts to their previous status

Thus, when a programmer invokes *kputc*, all processing stops until the character has been displayed. Once the character has been displayed, processing resumes. The important idea is that the operating system itself does not need to be working because *kputc* manipulates the hardware device directly.

Once *kputc* is available, it is easy to create a function that can display formatted output. Again following Unix tradition, we call the function *kprintf*. Basically, *kprintf* operates exactly like *printf* except that instead of invoking *putc* to display each character, *kprintf* invokes *kputc*.†

Although it is not important to understand the exact details of how polled I/O operates, it is essential to use polled I/O when debugging:

> *Whenever they modify or extend the operating system, programmers should use* kprintf *to display messages rather than* printf.

2.12 Perspective

Concurrent processing is one of the most powerful abstractions in computer science. It makes programming easier, less error prone, and in many cases, yields higher overall performance than code that attempts to switch among tasks manually. The advantages are so significant that once concurrent execution was introduced, it rapidly became the primary choice for most programming.

2.13 Summary

An understanding of an operating system begins with the set of services the system provides to applications. Unlike a conventional, sequential programming environment, an operating system provides concurrent execution in which multiple processes proceed at the same time. In our example system, as in most systems, a process can be created or terminated at runtime. Multiple processes can each execute a separate function, or multiple processes can execute a single function. In a concurrent environment, storage for arguments, local variables, and a function call stack are associated with each process rather than with the code.

---

†Debugging operating system code is difficult because disabling interrupts can change the execution of a system (e.g., by preventing clock interrupts). Thus, a programmer must be extremely careful when using *kprintf*.

Processes use synchronization primitives, such as semaphores, to coordinate execution. Two primary forms of coordination are producer–consumer synchronization and mutual exclusion.

EXERCISES

2.1 What is the API an operating system provides, and how is the API defined?

2.2 To what does *multitasking* refer?

2.3 List the two basic categories of multitasking systems, and state the characteristics of each.

2.4 What characteristics are generally associated with the terms *process*, *task*, and *thread*?

2.5 How is a process ID used?

2.6 How does calling function *X* differ from calling *create* to start a process executing function *X*?

2.7 The program in file *ex3.c* uses three processes. Modify the code to achieve the same results using only two processes.

2.8 Test the program in file *ex4.c* repeatedly. Does it always print the same number of zeroes? Does it ever print a value of *n* other than 0 or 2000?

2.9 In Xinu, what is the difference in storage between global variables and local variables?

2.10 Why do programmers avoid busy waiting?

2.11 Suppose three processes attempt to use function *additem* in file *ex6.c* at the same time. Explain the series of steps that occur, and give the value of the semaphore during each step.

2.12 Modify the producer–consumer code in file *ex5.c* to use a buffer of 15 slots, and have the producer and consumer synchronize in such a way that a producer can generate up to 15 values before blocking and a consumer can extract all values in the buffer before blocking. That is, arrange for the producer to write integers 1, 2, ... in successive locations of the buffer, wrapping around to the beginning after filling the last slot, and have the consumer extract values and print them on the console. How many semaphores are needed?

2.13 In file *ex5.c*, the semaphore *produced* is created with a count of 1. Rewrite the code so *produced* is created with a count of 0 and the producer signals the semaphore once before starting the iteration. Does the change affect the output?

2.14 Find the documentation for the serial port (or console device hardware) on a platform to which you have access. Describe how to construct a polled I/O function, *kputc()* that uses the device.

Chapter Contents

3.1 Introduction, 37
3.2 Physical And Logical Organizations Of A Platform, 38
3.3 Instruction Sets, 38
3.4 General-purpose Registers, 39
3.5 I/O Buses And The Fetch-Store Paradigm, 41
3.6 Direct Memory Access, 42
3.7 The Bus Address Space, 42
3.8 Bus Startup And Configuration, 43
3.9 Calling Conventions And The Runtime Stack, 44
3.10 Interrupts And Interrupt Processing, 47
3.11 Vectored Interrupts, 48
3.12 Exception Vectors And Exception Processing, 48
3.13 Clock Hardware, 49
3.14 Serial Communication, 49
3.15 Polled vs. Interrupt-driven I/O, 49
3.16 Storage Layout, 50
3.17 Memory Protection, 51
3.18 Hardware Details And A System On Chip Architecture, 51
3.19 Perspective, 52
3.20 Hardware References, 52

3

An Overview Of The Hardware And Runtime Environment

One machine can do the work of fifty ordinary men. No machine can do the work of one extraordinary man.

— Elbert Hubbard

3.1 Introduction

Because it deals with the details of devices, processors, and memory, an operating system cannot be designed without knowledge of the capabilities and features of the underlying hardware. Throughout the text, we will use two example hardware platforms: a *Galileo* and a *BeagleBone Black*. The platforms each consist of a small, low-cost circuit board that includes a processor, memory, and a few I/O devices. The processors on the boards employ well-known instruction sets: the Galileo uses Intel's instruction set that is known as *x86*, and the BeagleBone Black uses an *ARM* instruction set. We will see that most operating system functions are identical on the two architectures. However, the use of two platforms will allow a reader to compare how low-level OS system functions, such as a context switch, differ between a CISC (Complex Instruction Set Computer) architecture and a RISC (Reduced Instruction Set Computer) architecture.

The remainder of the chapter provides an overview of the two hardware platforms, describing pertinent features of the processors, memories, and I/O devices. The chapter

explains enough of the architecture, memory addressing, runtime stack, interrupt mechanisms, and device addressing to allow a reader to understand the discussions that follow. Although the chapter focuses on specific boards, the basic concepts apply broadly to most computer systems.

3.2 Physical And Logical Organizations Of A Platform

Physically, the platforms we are using each consist of a single, self-contained circuit board that uses a separate power cube. Most of the major components are contained in a single VLSI chip that is known as a *System on Chip* (*SoC*).† Because the boards are intended for use in experimental systems, they come completely assembled.

Like most computers, our platforms each include a *serial console* — a character-oriented device that the operating system uses to issue information messages, report errors, and interact with a user.‡ The Galileo has a (nonstandard) connector for its serial console. Thus, one must obtain a special cable. Although it has serial hardware on the board, the BeagleBone Black does not have a connector. Instead, a user must purchase a converter that connects to GPIO pins on the board. To interact over a serial console, a user can connect the serial line through a cable to a conventional computer (e.g., to the USB connection to a laptop or desktop), and then run an application that sends characters to the console line and displays the results. Information about a serial connection and instructions describing how to attach a serial connector can be found on the web site:

http://www.xinu.cs.purdue.edu

For now, it is sufficient to know that once an operating system is loaded on one of the platforms, a user can run an application that sends keystrokes to the platform and displays output from the platform.

Logically, the platforms follow the same overall architecture as most general-purpose computer systems. The components on the SoC include a processor, memory interface, and I/O device interfaces. Among other devices, each board includes an Ethernet network interface that allows the board to connect to a local network or the Internet. Figure 3.1 illustrates the conceptual organization.

3.3 Instruction Sets

Recall from the above discussion that the Galileo board implements an Intel instruction set and the BeagleBone Black implements an ARM instruction set. Except for a few special cases, an operating system does not need to focus on the instruction set because most OS functions are written in C, and a compiler generates the appropriate code. Thus, instruction set details will only be important for a handful of functions that are written in assembly language.

---

†See page 51 for an explanation of the difference between the boards and the underlying SoC.

‡Technically, a serial console uses the RS-232 standard.

Figure 3.1 The logical organization of major components in the example platforms.

In practice, the platforms each include additional hardware features that are not shown in the figure. For example, the BeagleBone Black provides graphics and floating point accelerators and an HDMI interface; the Galileo provides a mini-PCI Express interface and an interface for a micro-SD card. However, we are only concerned with the hardware features most pertinent to an operating system. For now, we will focus on the overall design of each component and how the components fit together. Later chapters discuss additional details, explain how an operating system interacts with the hardware, and provide examples.

3.4 General-purpose Registers

Most computer architectures provide a small set of general-purpose registers. We think of registers as temporary, high-speed storage that can be used in arithmetic, logical, and data movement instructions. A register can hold an integer, a pointer, or other data value. Because registers are much faster than memory, a compiler keeps frequently used variables in registers, and only writes values back to memory when the register is needed for another variable.

Hardware registers play a key role in operating system design because they form part of the state of a computation. That is, the hardware provides a single set of general-purpose registers, and each computation uses the registers. To support concurrent processes, an operating system must provide each process with the illusion that it has complete control over the registers. From a process's point of view, once the process can store a value in register X, the value will remain until the process changes it. We know that an operating system switches the processor among multiple processes. To preserve the illusion when switching from one process to another, the operating system must save a copy of the values that the first process had placed in all the registers, and load the registers with the values the second process was using when it last had

control of the processor. In Chapter 5, we will see exactly how an operating system saves and restores copies of registers. For now, it is sufficient to understand that the hardware contains one copy of registers that all processes share.

3.4.1 Galileo (Intel)

The Galileo board follows the traditional 32-bit Intel architecture. In addition to an internal *program counter* and floating point registers, the processor has eight general-purpose registers. For our purposes, each register is 32 bits wide.† Figure 3.2 lists the registers on an Intel processor by giving the name used to reference a register and the meaning assigned.

Name	Use
EAX	Accumulator
EBX	Base
ECX	Count
EDX	Data
ESI	Source Index
EDI	Destination Index
EBP	Base Pointer
ESP	Stack Pointer

Figure 3.2 The general-purpose registers in the Galileo board (Intel) and the meaning of each.

In practice, the Intel instruction set allows a programmer to access subparts of registers. For example, the hardware allows a programmer to treat some of the 32-bit registers as pairs of 16-bit registers. That is, instructions can access the low-order 16 bits or the high-order 16 bits of a register without changing the other part. In fact, some instructions allow a programmer to reference individual bytes of a register. However, such details will not be important except in special cases, such as system startup.

Registers can be used as operands in instructions, or can point to an operand in memory. Despite being classified as general-purpose, individual registers are assigned specific uses by the hardware and/or compiler. For example, the *ESP* register serves as a pointer to a runtime stack. The stack can be used as temporary storage during computation (e.g., by pushing intermediate results and popping them later), and is essential during function invocation because the activation record for a called function is pushed on the stack during a function call, and popped off the stack when the function returns.

†Xinu runs in *protected mode*; an x86 processor has other modes, such as a 64-bit mode, with more general-purpose registers.

3.4.2 BeagleBone Black (ARM)

The BeagleBone Black follows the traditional 32-bit ARM architecture, which has 15 general-purpose registers plus a *program counter*. The program counter contains the address of the instruction that will be executed next, and is only changed during a jump. Figure 3.3 lists the 32-bit registers on an ARM processor, aliases, and their typical use.†

Name	Alias	Use
R0 – R3	a1 – a4	Argument registers
R4 – R11	v1 – v8	Variables and temporaries
R9	sb	Static base register
R12	ip	Intra procedure call scratch register
R13	sp	Stack pointer
R14	lr	Link register used for return address
R15	pc	Program counter

Figure 3.3 The general-purpose registers and the program counter in the BeagleBone Black (ARM) and the meaning of each.

3.5 I/O Buses And The Fetch-Store Paradigm

We use the term *bus* to refer to a mechanism that provides the primary path between the processor and other components, namely the memory, I/O devices, and other interface controllers. Bus hardware uses a *fetch-store paradigm* in which the only operations are *fetch*, to move data from a component across the bus to the processor, and *store*, to move data from the processor across the bus to a component. For example, when it needs to access memory, the processor places a memory address on the bus and issues a *fetch* request to obtain the corresponding value. The memory hardware responds to the request by looking up the address in memory, placing the data value on the bus, and signalling the processor that the value is ready. Similarly, to store a value in memory, the processor places an address and value on the bus and issues a *store* request; the memory hardware extracts the value and stores a copy in the specified memory location. Bus hardware handles many details of the fetch-store paradigm, including signals that the processor and other components use to communicate and control access to the bus. We will see that an operating system can use a bus without knowing many details of the underlying hardware.

The example systems use *memory-mapped I/O*, which means that each I/O device is assigned a set of addresses in the bus address space. The processor uses the same

†The ARM architecture has eight additional floating point registers that are not listed here because they are not directly relevant to process management.

fetch-store paradigm to communicate with I/O devices as with memory. We will see that communication with a memory-mapped I/O device resembles data access. First, the processor computes the address associated with a device. Second, to access the device, the processor either stores a value to the address or fetches a value from the address.

3.6 Direct Memory Access

Higher-speed I/O devices (e.g., an Ethernet device) offer *Direct Memory Access* (*DMA*), which means the device contains hardware that can use the bus to communicate directly with memory. The key idea is that DMA allows I/O to proceed quickly because it does not interrupt the processor frequently nor does it require the processor to perform each data transfer. Instead, a processor can give the I/O device a list of operations, and the device proceeds from one to the next. Thus, DMA allows a processor to continue running processes while a device operates.

As an example, consider how an Ethernet device uses DMA. To receive a packet, the operating system allocates a buffer in memory and starts the Ethernet device. When a packet arrives, the device hardware accepts the packet and makes multiple bus transfers to move a copy into the buffer in memory. Finally, once the entire packet has been transferred, the device interrupts the processor. Sending a packet is equally efficient: the operating system places the packet in a buffer in memory and starts the device. The device uses the bus multiple times to fetch the packet from the buffer in memory. The device then transmits the packet on the network and interrupts the processor after the entire packet has been transferred.

We will see that the DMA hardware on the example platforms allows a processor to request multiple operations. In essence, the processor creates a list of packets to be sent and a list of buffers to be used for incoming packets. The network interface hardware uses the lists to send and receive packets without requiring the processor to restart the device after each operation. As long as the processor consumes incoming packets faster than they arrive and adds fresh buffers to the list, the network hardware device will continue to read packets. Similarly, as long as the processor continues to generate packets and add them to the list, the network hardware device will continue to transmit the packets. Later chapters explain additional DMA details, and the example code illustrates how a device driver in the operating system allocates I/O buffers and controls DMA operations.

3.7 The Bus Address Space

Each of the platforms uses a 32-bit bus address space, with addresses ranging from 0x00000000 through 0xFFFFFFFF. Some of the addresses in the bus address space correspond to memory, some to FlashROM, and others to I/O devices. The next sections provide more detail.

Memory. On each system, memory is divided into 8-bit *bytes*, with a byte being the smallest addressable unit. The C language uses the term *character* in place of *byte* because each byte can hold one ASCII character. Although a 32-bit bus can address 4 Gbytes total, not all possible addresses are assigned. The Galileo contains an 8 Mbyte Legacy SPI Flash (used to store firmware, such as a bootstrap loader), 512 Kbytes of SRAM, 256 Mbytes of DRAM, and between 256 and 512 Kbytes of storage for Arduino sketches. Note that the largest item, the DRAM, occupies only 6.25% of the address space. The BeagleBone Black contains a 4 Gbyte Flash memory (used to store firmware), and 512 Mbytes of DRAM. In the case of a BeagleBone Black, the DRAM occupies 12.5% of the bus address space, meaning that many addresses are not assigned.

Do unassigned bus addresses cause a problem? The hardware allows addresses to remain unassigned provided the addresses are not referenced. If the processor attempts to access an unassigned address, however, the hardware raises an *exception*.† For example, a bus exception can occur if a Xinu process overflows an array or generates an incorrect pointer and then tries to dereference the pointer. The important point is:

Referencing an unassigned bus address, such as an address beyond the physical memory, will cause the hardware to create an exception.

3.8 Bus Startup And Configuration

How should memory modules and devices be assigned addresses in the bus address space? Two methods are used:

- Static address assignment
- Dynamic address assignment

The example platforms illustrate the two methods: the BeagleBone Black uses static assignment, and the Galileo uses dynamic assignment.

Static address assignment. Often used by small, embedded systems, static means that all details of the hardware configuration are chosen when the hardware is designed. That is, a hardware designer chooses a set of peripheral devices, and assigns each a unique address on the bus. The hardware designer passes the assignments to an operating system designer, who must configure the operating system to match the underlying hardware. Extreme care must be taken — if the operating system does not match the hardware exactly, the system will not operate correctly (i.e., may not even be able to boot). Static configuration has the advantage of increased runtime efficiency, but the disadvantages of less generality and susceptibility to human error.

Dynamic address assignment. Dynamic assignment means that when a system boots, an operating system, with the support of the bus hardware and platform firmware, must discover devices and memories attached to the bus. Each component attached to the bus is assigned a unique identifier that specifies the vendor and the specific

†Chapter 24 discusses exception handling.

hardware type. An operating system probes the bus to determine which components are present. Each component returns its ID. For example, an operating system can determine the type and size of each memory module that has been installed as well as the exact bus address assigned to each module. A bus that supports dynamic assignment has the advantage of increased generality because a single operating system image can run on a variety of hardware configurations. However, dynamic assignment makes both bus hardware and operating system code more complex.

Chapter 25 discusses operating system configuration in more detail, and shows an example of configuration; chapters on device mechanisms and drivers show how a system contends with static and dynamic address assignment.

3.9 Calling Conventions And The Runtime Stack

We will see that operating systems rely heavily on mechanisms for function invocation. When they invoke operating systems services, applications use function calls. Thus, function calls define operating system services. Furthermore, when it switches context from one process to another, an operating system must handle the situation where the processes have each made a series of nested function calls. Thus, an operating system designer must understand the details of function invocation. The following paragraphs define key concepts and mechanisms related to function invocation.

Calling conventions. The steps taken during a function call and return are known as *calling conventions*. The term *convention* arises because the hardware does not dictate all details. Instead, the hardware design places some constraints on possible approaches, and leaves final choices to a compiler writer. We will see that because it calls functions to process interrupts and switch from one running process to another, an operating system must understand and follow the same conventions as the compiler.

Arguments and argument passing. When a function is called, the caller supplies a set of actual *arguments* that correspond to formal *parameters*. A variety of argument passing mechanisms have been used in commercial processors. The example platforms illustrate two common approaches: placing arguments on the stack and passing arguments in general-purpose registers.

Runtime stack and stack frame contents. A statically scoped language, such as C, uses a *runtime stack* to store the state associated with a function call. The compiler allocates enough space on the stack to hold an *activation record* for the called function. The allocated space is known as a *stack frame*. Each activation record contains space for the local variables associated with the function, temporary storage needed during computation, a return address, and other miscellaneous items. We assume the stack grows downward from higher memory addresses to lower memory addresses. A compiler generates code needed to create a stack frame. The compiler computes the size needed for local variables as well as the size needed for auxiliary items, such as the temporary space used to save the contents of registers (so the function can use the registers for computation and then restore their original values before returning to the caller). We will see that when it creates a new process, an operating system needs to construct a

stack frame for the process's initial function. Thus, the operating system needs to know exactly how a stack frame is laid out, including the location of arguments and the return address. Examples will clarify the concept. In the examples below, we assume a *gcc* compiler; the calling conventions used by other compilers may differ in minor ways.

3.9.1 Galileo (Intel)

On an Intel processor, before invoking a function, a caller pushes registers *EAX*, *ECX*, and *EDX* on the stack, pushes the arguments in reverse order, and then executes a *call* instruction, which pushes the return address on the stack. Code in the called function pushes *EBP*, *EBX, EDI*, and *ESI*, and pushes space for the local variables. When the call returns, values are popped from the stack in reverse order. Figure 3.4 illustrates values on the top of the stack immediately after a function call (remember that a stack grows downward).

Figure 3.4 The stack on an Intel processor when a function is called.

3.9.2 BeagleBone Black (ARM)

On an ARM processor, the first four arguments to a function are passed in registers *a0* – *a3* (i.e., registers *r0* – *r4*). Arguments beyond the first four are passed on the stack. To call a function, the caller executes a *BL* (branch and link) instruction. When the BL executes, the hardware places the return address in register *r14*. The called function must save registers it will use; typically, the called function pushes copies of *r14* – *r4* onto the stack, and then pushes the *CPSR* (status) register. Figure 3.5 illustrates values on the top of the stack immediately after a function call (remember that a stack grows downward). In the figure, the top of the stack only contains space for local variables beyond the first seven. By convention, the first seven local variables are stored in registers 4 through 8, 10, and 11.

In fact, the set of saved registers varies across versions of the ARM architecture. Furthermore, if a program uses floating point, the floating point registers must also be saved and restored.

Figure 3.5 The stack on an ARM processor when a function is called.

3.10 Interrupts And Interrupt Processing

Modern processors provide mechanisms that allow external I/O devices to *interrupt* the processor when they need service. In most cases, processor hardware has a closely related *exception* mechanism that is used to inform the software when an error or fault occurs (e.g., an application attempts division by zero or references a page in virtual memory that is not present in memory). From an operating system's point of view, interrupts are fundamental because they allow the processor to perform computation at the same time I/O proceeds.†

Any of the I/O devices connected to a bus can interrupt the processor when the device needs service. To do so, the device places a signal on one of the bus control lines. During normal execution of the fetch-execute cycle, hardware in the processor monitors the control line and initiates interrupt processing when the control line has been signaled. In the case of a RISC processor, the main processor does not usually contain the hardware to handle interrupts. Instead, a co-processor interacts with the bus on behalf of the main processor.

Whatever interrupt mechanisms a processor uses, the hardware or the operating system must guarantee that:

- The entire state of the processor, including the program counter and status registers, is saved when an interrupt occurs
- The processor runs the appropriate interrupt handler processor, which must have been placed in memory before the interrupt occurs
- When an interrupt finishes, the operating system and hardware provide mechanisms that restore the entire state of the processor and continue processing at the point of interruption

Interrupts introduce a fundamental idea that pervades an entire operating system. An interrupt can occur at any time, and an operating system can switch from one process to another during an interrupt. The consequence is that other processes can run at any time.

To prevent problems caused by concurrent processes trying to manipulate shared data, an operating system must take steps to avoid switching context. The simplest way to prevent other processes from executing consists of disabling interrupts. That is, the hardware includes an *interrupt mask* mechanism that an operating system can use to control interrupts. On many hardware systems, if the interrupt mask is assigned a value of zero, the processor ignores all interrupts; if the mask is assigned a non zero value, the hardware allows interrupts to occur. On some processors, the hardware has individual interrupt bits for each device, and on others, the mask provides a set of eight or sixteen levels, and each device is assigned a level. We will see that many operating system functions disable interrupts while they manipulate global data structures and I/O queues.

---

†Later chapters explain how an operating system manages interrupt and exception processing, and show how the high-level I/O operations a user performs relate to low-level device hardware mechanisms.

3.11 Vectored Interrupts

When a device interrupts, how does the hardware know the location of the code that handles the interrupt? The hardware on most processors uses a mechanism known as *vectored interrupts*. The basic idea is straightforward: each device is assigned a small integer number: 0, 1, 2, and so on. The integers are known as *interrupt level numbers* or *interrupt request numbers*. The operating system creates an array of pointers in memory known as an *interrupt vector*, where the ith entry in the interrupt vector array points to the code that handles interrupts for the device with vector number i. When it interrupts, a device sends its vector number over the bus to the processor. Depending on the processor details, either the hardware or the operating system uses the vector number as an index into the interrupt vector, obtains a pointer, and uses the pointer as the address of the code to run.

Because it must be configured before any interrupts occur, an operating system initializes the interrupt vector at the same time devices are assigned addresses on the bus. The assignment of interrupt level numbers usually employs the same paradigm as address assignment. A manual assignment means a human assigns a unique interrupt level number to each device and then configures the interrupt vector addresses accordingly. An automatic approach requires bus and device hardware that can assign interrupt levels at runtime. To use the automatic approach, an operating system polls devices at startup, assigns a unique interrupt level number to each device, and initializes the interrupt vector accordingly. Automatic assignment is safer (i.e., less prone to human error), but requires more complex hardware in both the devices and the bus. We will see examples of static and automatic interrupt vector assignment.

3.12 Exception Vectors And Exception Processing

Many processors follow the same vectored approach for *exceptions* as they use for interrupts. That is, each exception is assigned a unique number: 0, 1, 2, and so on. When an exception occurs, the hardware places the exception number in a register. The operating system extracts the exception number, and uses the number as the index into an *exception vector*. A minor difference occurs between the way processor hardware handles interrupts and exceptions. We think of an interrupt as occurring *between* two instructions. Thus, one instruction has completed and the next instruction has not begun. However, an exception occurs *during* an instruction. Thus, when the processor returns from the exception, the program counter has not advanced, and the instruction can be restarted. Restarting is especially important for page fault exceptions — when a page fault occurs, the operating system must read the missing page from memory, set the page table, and then execute the instruction that caused the fault a second time.

3.13 Clock Hardware

In addition to I/O devices that transfer data, most computers include hardware that can be used to manage timed events. There are two basic forms:

- Real-time clock
- Interval timer

Real-time clock. A real-time clock circuit consists of hardware that generates a pulse regularly (e.g., 1000 times per second). To turn a real-time clock circuit into a real-time clock device, the hardware is configured to interrupt the processor on each pulse. A real-time clock device does not keep any counters, does not store the time of day, and may not have an adjustable cycle time (e.g., the pulse rate may be determined by a crystal that must be replaced to change the rate).

Interval timer. Conceptually, an interval timer consists of a real-time clock circuit that pulses at regular intervals connected to a counter that computes a tally of pulses plus a comparator circuit that compares the tally to a threshold value. An operating system can specify a threshold value and can reset the counter to zero. When the counter reaches the threshold value, the interval timer interrupts the processor. The advantage of an interval timer lies in its efficiency. Instead of interrupting continuously, an interval timer can be configured to wait until an event should occur. Of course, interval timer hardware is more complex than a real-time clock.

3.14 Serial Communication

Serial communication devices are among the simplest I/O devices available, and have been used on computers for decades. Each of the example platforms contains an RS-232 serial communication device that is used as a system console. The serial hardware handles both input and output (i.e., the transmission and reception of characters). When an interrupt occurs, the processor examines a device hardware register to determine whether the output side has completed transmission or the input side has received a character. Chapter 15 examines serial devices, and shows how interrupts are processed.

3.15 Polled vs. Interrupt-driven I/O

Most I/O performed by an operating system uses the interrupt mechanism. The operating system interacts with the device to start an operation (either input or output), and then proceeds with computation. When the I/O operation completes, the device interrupts the processor, and the operating system can choose to start another operation.

Although they optimize concurrency and permit multiple devices to proceed in parallel with computation, interrupts cannot always be used to perform I/O. For exam-

ple, consider displaying a startup message for a user before the operating system has initialized interrupts and I/O. Also consider a programmer writing operating systems code. It may be desirable to allow I/O even though it may be necessary to leave interrupts disabled during debugging. In either case, interrupts cannot be used.

The alternative to *interrupt-driven I/O* is known as *polled I/O*. When using polled I/O, the processor starts an I/O operation, but does not enable interrupts. Instead, the processor enters a loop that repeatedly checks a device status register to determine whether the operation has completed. We have already seen an example of how an operating system designer can use polled I/O when we examined functions *kputc* and *kprintf* in Chapter 2.

3.16 Storage Layout

When it compiles a program, a C compiler partitions the resulting image into four memory segments:

- Text segment
- Data segment
- Bss segment
- Stack segment

The *text segment*, which includes code for the main program and all functions, occupies the lowest part of the address space. The *data segment*, which contains all initialized data, occupies the next region of the address space. The uninitialized data segment, called the *bss segment*, follows the data segment. Finally, the *stack segment* occupies the highest part of the address space and grows downward. Figure 3.6 illustrates the conceptual organization.

Figure 3.6 Illustration of memory segments created by a C compiler.

The symbols *etext*, *edata*, and *end* in the figure refer to global variables that the loader inserts into the object program. The names are initialized to the first address beyond the text, data, and bss segments, respectively. Thus, a running program can determine how much memory remains between the end of the bss segment and the current top of stack by subtracting the address of *end* from the stack pointer, SP.

Chapter 9 explains memory allocation for multiple processes in Xinu. Although all processes share the text, data, and bss segments, a separate stack segment must be allo-

cated for each process. If three processes are executing, the stacks are allocated contiguously from the highest memory address downward as Figure 3.7 illustrates.

As the figure indicates, each process has its own stack pointer. At a given time, the stack pointer for process *i* must point to an address within the stack that has been allocated for process *i*. Later chapters explain the concept in detail.

Figure 3.7 Illustration of memory with stack segments for three processes.

3.17 Memory Protection

The memory hardware available on the example platforms includes mechanisms that an operating system can use to protect one process from another and to protect the operating system from application processes. For example, applications can be configured to run in *user mode*, which means they cannot read or write kernel memory. When an application makes a system call, control transfers to the kernel and the privilege level is increased to *kernel mode* until the call returns. The key to understanding protection is to remember that control can only transfer to the operating system at the specific entry points that the operating system designer provides. Thus, a designer can ensure that an application only receives carefully controlled services.

Like many other embedded systems, our example code avoids the complexity and runtime overhead of memory protection. Instead the code runs completely in a privileged mode, without memory protection. The lack of protection means a programmer must be careful because any process can access any memory location, including the memory allocated to operating system structures or memory allocated to another process's stack. If a process overflows the allocated stack area, the process's runtime stack will overwrite data in another process's stack.

3.18 Hardware Details And A System On Chip Architecture

Both hardware platforms use a *System on Chip* (*SoC*) approach. That is, a single VLSI chip contains the processor, memories, and a set of I/O interfaces. For example, the Intel Galileo contains a *Quark* SoC. We have chosen to emphasize the boards rather than the underlying SoC because a given board can be less general than the SoC. For example, the Quark design allows multiple cores and has multiple copies of some I/O interfaces. However, the version of the SoC on the Galileo board provides only one core and one Ethernet connection. Similarly, the ARM architecture is more general than the version available on the BeagleBone Black.

Despite our focus on complete boards rather than the SoC, much of the material in the text generalizes to other boards that use the same SoC. For example, the context switch code used with the Galileo also runs on other x86 platforms, and the context switch code used on the BeagleBone Black also runs on other ARM platforms.

3.19 Perspective

The hardware specifications for a processor or I/O device contain so many details that studying them can seem overwhelming. Fortunately, many of the differences among processors are superficial — fundamental concepts apply across most hardware platforms. Therefore, when learning about hardware, it is important to focus on the overall architecture and design principles rather than on tiny details.

In terms of operating systems, many hardware details affect the overall design. In particular, the hardware interrupt mechanism and interrupt processing dominate many parts of the design. Intellectually, however, the most significant item to appreciate is the stunning disparity between the primitive facilities the hardware offers and the high-level abstractions an operating system supplies.

3.20 Hardware References

General information about the BeagleBone Black, including a photo and links to a *Getting Started* page and other materials can be found on the web site:

http://beagleboard.org/black

Details about the processor and SoC can be found on:

http://www.ti.com/product/am3358

and a data sheet for the SoC is available at:

http://www.ti.com/lit/pdf/spruh73

A general introduction to the Galileo and other platforms that use the Quark SoC can be found at:

http://www.intel.com/galileo

Information about the Galileo development board, including a photo and links to sites with more detail, can be found at:

http://www.intel.com/content/www/us/en/intelligent-systems/galileo/galileo-overview.html

EXERCISES

3.1 Some systems use a *programmable* interrupt address mechanism that allows the system to choose the address to which the processor jumps when an interrupt occurs. What is the advantage of a programmable interrupt address?

3.2 DMA introduces the possibility of unexpected errors. What happens if a DMA operation that transfers N bytes of data begins at a memory location less than N bytes from the highest memory address?

3.3 Read about hardware that uses multi-level interrupts. Should an interrupt at one level be able to interrupt the system while it is processing an interrupt from another level? Explain.

3.4 What are the advantages of the memory layout shown in Figure 3.7? Are there disadvantages? What other layouts might be useful?

3.5 Embedded hardware often includes multiple interval timer devices, each with its own interrupt vector. Why might multiple timers be helpful? Can a system with only a single timer accomplish the same tasks as a system with multiple timers? Explain.

3.6 If you are familiar with an assembly language, read about the calling conventions that are used to permit recursive function calls. Build a function that makes recursive calls, and demonstrate that your function works correctly.

3.7 On an Intel platform, the program counter (i.e., the instruction pointer register) cannot be read directly. Devise x86 assembly language code that determines the value indirectly by using instructions that manipulate the instruction pointer.

Chapter Contents

4.1 Introduction, 57
4.2 A Unified Structure For Linked Lists Of Processes, 58
4.3 A Compact List Data Structure, 59
4.4 Implementation Of The Queue Data Structure, 61
4.5 Inline Queue Manipulation Functions, 62
4.6 Basic Functions To Extract A Process From A List, 63
4.7 FIFO Queue Manipulation, 65
4.8 Manipulation Of Priority Queues, 68
4.9 List Initialization, 70
4.10 Perspective, 71
4.11 Summary, 72

4

List And Queue Manipulation

As some day it may happen that a victim must be found, I've got a little list ....

— W. S. Gilbert

4.1 Introduction

Linked list processing is fundamental in operating systems, and pervades each component. Linked structures enable a system to manage sets of objects efficiently without searching or copying. As we will see, managing lists of processes is especially important.

This chapter introduces a set of functions that form the backbone of a linked list manipulation system. The functions represent a unified approach — a single data structure and a single set of nodes used by all levels of the operating system to maintain lists of processes. We will see that the data structure includes functions to create a new list, insert an item at the tail of a list, insert an item in an ordered list, remove the item at the head of a list, or remove an item from the middle of a list.†

The linked list functions are easy to understand because they assume that only one process executes a list function at a given time. Thus, a reader can think of the code as being part of a sequential program — there is no need to worry about interference from other processes executing concurrently. In addition, the example code introduces several programming conventions used throughout the text.

†Although linked list manipulation is usually covered in texts on data structures, the topic is included here because the data structure is unusual and because it forms a key part of the system.

4.2 A Unified Structure For Linked Lists Of Processes

A process manager handles objects called *processes*. Although at any time a process appears on only one list, a process manager moves a process from one list to another frequently. In fact, a process manager does not store all details about a process on a list. Instead, the process manager merely stores a process ID, a small, nonnegative integer used to reference the process. Because it is convenient to think of placing a process on a list, we will use the terms *process* and *process ID* interchangeably throughout the chapter.

An early version of Xinu had many lists of processes, each with its own data structure. Some consisted of first-in-first-out (FIFO) queues, and others were ordered by a key. Some lists were singly linked; others needed to be doubly linked to permit items to be inserted and deleted at arbitrary positions efficiently. After the requirements had been formulated, it became clear that centralizing the linked-list processing into a single data structure would reduce code size and eliminate many special cases. That is, instead of six separate sets of linked list manipulation functions, a single set of functions was created to handle all situations.

To accommodate all cases, a representation was selected with the following properties:

- All lists are doubly linked, which means a node points to its predecessor and successor.
- Each node stores a key as well as a process ID, even though a key is not used in a FIFO list.
- Each list has head and tail nodes; the head and tail nodes have the same memory layout as other nodes.
- Non-FIFO lists are ordered in descending order; the key in a head node is greater than the maximum valid key value, and the key value in the tail node is less than the minimum valid key.

Figure 4.1 illustrates the conceptual organization of a linked list data structure by showing an example list with two items.

Figure 4.1 The conceptual organization of a doubly-linked list containing processes 4 and 2 with keys 25 and 14, respectively.

As expected, the successor of the tail and the predecessor of the head are null. When a list is empty, the successor of the head is the tail and the predecessor of the tail is the head, as Figure 4.2 illustrates.

Figure 4.2 The conceptual form of an empty linked list.

4.3 A Compact List Data Structure

One of the key design goals in an embedded system involves reducing the memory used. Instead of using a conventional implementation of a linked list, Xinu optimizes the memory required in two ways:

- Relative pointers
- Implicit data structure

To understand the optimizations, it is important to know that most operating systems place a fixed upper bound on the number of processes in the system. In Xinu, constant *NPROC* specifies the number of processes, and process identifiers range from *0* through *NPROC – 1*. In most embedded systems, *NPROC* is small (less than 100); we will see that a small limit makes each optimization work well.

Relative pointers. To understand the motivation for relative pointers, consider the space a conventional pointer occupies. On a 32-bit architecture, each pointer occupies four bytes. If the system contains fewer than 100 nodes, however, the size required can be reduced by placing nodes in contiguous memory locations and using a value between 0 and 99 as a reference. That is, the nodes can be allocated in an array, and the array index can be used instead of a pointer.

Implicit data structure. The second optimization focuses on omitting the process ID field from all nodes. Such an omission is feasible because:

A process appears on at most one list at any time.

To omit the process ID, use an array implementation and use the ith element of the array for process ID i. Thus, to put process 3 on a particular linked list, insert node 3 onto the list. Thus, the relative address of a node is the same as the ID of the process being stored.

Figure 4.3 illustrates how the linked list in Figure 4.1 can be represented in an array that incorporates relative pointers and implicit identifiers. Each entry in an array has three fields: a key, the index of the previous node, and the index of the next node. The head of the list has index 60, and the tail has index 61.

Figure 4.3 The list from Figure 4.1 stored in the queue table array.

Because a *NEXT* or *PREV* field contains a relative pointer (i.e., an array index), the size of the field depends on the size of the array. For example, if the array contains fewer than 256 items, a single byte can be used to store an index.

Xinu uses the term *queue table* to refer to the array. The key to understanding the structure is to observe that array elements with an index less than *NPROC* differ from elements with a higher index. Positions 0 through *NPROC–1* each correspond to one process in the system; positions *NPROC* and higher are used to hold head and tail pointers for lists. Such a data structure is only feasible because the maximum number of processes and the maximum number of lists are each known at compile time and a process can only appear on one list at a given time.

4.4 Implementation Of The Queue Data Structure

To place process *i* on a list, the node with index *i* is linked into the list. A closer look at the code will make the operations clear. In Xinu, the queue table pictured above is named *queuetab*, and declared to be an array of *qentry* structures. File *queue.h* contains the declarations of both *queuetab* and *qentry*:

```
/* queue.h - firstid, firstkey, isempty, lastkey, nonempty              */

/* Queue structure declarations, constants, and inline functions        */

/* Default # of queue entries: 1 per process plus 2 for ready list plus */
/*                      2 for sleep list plus 2 per semaphore           */
#ifndef NQENT
#define NQENT   (NPROC + 4 + NSEM + NSEM)
#endif

#define EMPTY   (-1)            /* Null value for qnext or qprev index  */
#define MAXKEY  0x7FFFFFFF      /* Max key that can be stored in queue  */
#define MINKEY  0x80000000      /* Min key that can be stored in queue  */

struct  qentry  {               /* One per process plus two per list    */
        int32   qkey;           /* Key on which the queue is ordered    */
        qid16   qnext;          /* Index of next process or tail        */
        qid16   qprev;          /* Index of previous process or head    */
};

extern  struct qentry   queuetab[];

/* Inline queue manipulation functions */

#define queuehead(q)    (q)
#define queuetail(q)    ((q) + 1)
#define firstid(q)      (queuetab[queuehead(q)].qnext)
#define lastid(q)       (queuetab[queuetail(q)].qprev)
#define isempty(q)      (firstid(q) >= NPROC)
#define nonempty(q)     (firstid(q) <  NPROC)
#define firstkey(q)     (queuetab[firstid(q)].qkey)
#define lastkey(q)      (queuetab[ lastid(q)].qkey)

/* Inline to check queue id assumes interrupts are disabled */

#define isbadqid(x)     (((int32)(x) < 0) || (int32)(x) >= NQENT-1)
```

The *queuetab* array contains *NQENT* entries. As Figure 4.3 indicates, an important implicit boundary occurs between element *NPROC–1* and element *NPROC*. Each element below the boundary corresponds to a process ID, and elements *queuetab[NPROC]* through *queuetab[NQENT]* correspond to the heads or tails of lists.

File *queue.h* introduces several features of C and conventions used throughout the book. Because the name ends in *.h*, the file will be included in other programs ("h" stands for *header*). Such files often contain the declarations for global data structures, symbolic constants, and inline functions (macros) that operate on the data structures. File *queue.h* defines *queuetab* to be an external variable (i.e., global), which means that every process will be able to access the array. The file also defines symbolic constants used with the data structure, such as constant *EMPTY* that is used to define an empty list.

Symbolic constant *NQENT*, which defines the total number of entries in the *queuetab* array, provides an example of conditional definition. The statement *#ifndef NQENT* means "compile the code down to the corresponding *#endif*, if and only if *NQENT* has not been defined previously." Thus, the code in *queue.h* assigns a value to *NQENT* only if it has not been defined previously. The value assigned,

$$NPROC + 4 + NSEM + NSEM$$

allocates enough entries in *queuetab* for *NPROC* processes plus head and tail pointers for *NSEM* semaphore lists, a ready list, and a sleep list. Conditional compilation is used to permit the size of the *queuetab* array to be changed without modifying the *.h* file.

The contents of entries in the *queuetab* array are defined by structure *qentry*. The file contains only a declaration of the elements in the *queuetab* array; Chapter 22 explains how data structures are initialized at system startup. Field *qnext* gives the relative address of the next node on a list, field *qprev* points to the previous node, and field *qkey* contains an integer key for the node. When a field, such as a forward or backward pointer, does not contain a valid index value, the field is assigned the value *EMPTY*.

4.5 Inline Queue Manipulation Functions

The functions *isempty* and *nonempty* are predicates (i.e., Boolean functions) that test whether a list is empty or not empty, given the index of its head as an argument. *Isempty* determines whether a list is empty by checking to see if the first node on the

list is a process or the list tail; function *nonempty* makes the opposite test. To understand the test, remember that the first *NPROC* entries in the *queuetab* array each correspond to a process. All the other nodes are either a list head or tail. Therefore, testing whether an item on a list is a process only requires comparing the item's index to *NPROC* — the node is a process if its index is less than *NPROC*. The important point to note is that the test for an empty or nonempty queue are extremely efficient.

The other inline functions are also easy to understand. Function *firstkey* returns the key of the first process on a list, and function *lastkey* returns the key of the last process on a list. Function *firstid* returns the ID of the first process on a list. That is, the function returns the *queuetab* index of the first process on a list.

Usually, the queue functions are applied to nonempty lists. However, the implementation has a useful property: a list always has a head and tail, even if the list is empty. Thus, the functions described above can be applied to an empty list without causing a runtime error. For example, when *firstkey* is applied to an empty list, the function returns the key of the node following the head, which will be the tail. Thus, the function will return the key from the tail, *MINKEY*. Similarly, extracting the ID of the first node in an empty list will never cause an array reference to be out-of-bounds because the tail node has a valid ID. Finally, extracting the first key of an empty list will never cause an error because the *qkey* field is always initialized, even in the head and tail nodes.

4.6 Basic Functions To Extract A Process From A List

Consider extracting a process from a list.† Recall that extracting an item from the head of a FIFO queue results in removing the item that has been in the queue the longest. For a priority queue, extracting from the head produces an item with highest priority. Similarly, extracting an item from the tail of the queue produces an item with lowest priority. As a result, we can construct three basic functions that are sufficient to handle extraction:

- *getfirst* — extract the process at the head of a list
- *getlast* — extract the process at the tail of a list
- *getitem* — extract a process at an arbitrary point

The code for the three basic functions can be found in file *getitem.c*.

†A later section considers inserting a process into a list.

```
/* getitem.c - getfirst, getlast, getitem */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  getfirst  -  Remove a process from the front of a queue
 *------------------------------------------------------------------------
 */
pid32   getfirst(
          qid16         q               /* ID of queue from which to    */
        )                               /* Remove a process (assumed    */
                                        /*   valid with no check)       */
{
        pid32   head;

        if (isempty(q)) {
                return EMPTY;
        }

        head = queuehead(q);
        return getitem(queuetab[head].qnext);
}

/*------------------------------------------------------------------------
 *  getlast  -  Remove a process from end of queue
 *------------------------------------------------------------------------
 */
pid32   getlast(
          qid16         q               /* ID of queue from which to    */
        )                               /* Remove a process (assumed    */
                                        /*   valid with no check)       */
{
        pid32 tail;

        if (isempty(q)) {
                return EMPTY;
        }

        tail = queuetail(q);
        return getitem(queuetab[tail].qprev);
}

/*------------------------------------------------------------------------
 *  getitem  -  Remove a process from an arbitrary point in a queue
 *------------------------------------------------------------------------
```

```
 */
pid32   getitem(
          pid32         pid             /* ID of process to remove      */
        )
{
        pid32   prev, next;

        next = queuetab[pid].qnext;     /* Following node in list       */
        prev = queuetab[pid].qprev;     /* Previous node in list        */
        queuetab[prev].qnext = next;
        queuetab[next].qprev = prev;
        return pid;
}
```

Getfirst takes a queue ID as an argument, verifies that the argument identifies a nonempty list, finds the process at the head of the list, and calls *getitem* to extract the process from the list. Similarly, *getlast* takes a queue ID as an argument, checks the argument, finds the process at the tail of the list, and calls *getitem* to extract the process. Each of the two functions returns the ID of the process that has been extracted.

Getitem takes a process ID as an argument, and extracts the process from the list in which the process is currently linked. Extraction consists of making the previous node point to the successor and the successor point to the previous node. Once a process has been unlinked from a list, *getitem* returns the process ID as the value of the function.

4.7 FIFO Queue Manipulation

We will see that many of the lists a process manager maintains consist of a *First-In-First-Out* (*FIFO*) queue. That is, a new item is inserted at the tail of the list, and an item is always removed from the head of the list. For example, a scheduler can use a FIFO queue to implement round-robin scheduling by placing the current process on the tail of a list and switching to the process on the head of the list.

Functions *enqueue* and *dequeue*, found in file *queue.c*, implement FIFO operations on a list. Because each list has both a head and tail, both insertion and extraction are efficient. For example, *enqueue* inserts a process just prior to the tail of a list, and *dequeue* extracts an item just after the head of the list. *Dequeue* takes a single argument that gives the ID of the list to use. *Enqueue* takes two arguments: the ID of the process to be inserted and the ID of a list on which to insert it.

```
/* queue.c - enqueue, dequeue */

#include <xinu.h>

struct qentry   queuetab[NQENT];        /* Table of process queues      */

/*------------------------------------------------------------------------
 *  enqueue  -  Insert a process at the tail of a queue
 *------------------------------------------------------------------------
 */
pid32   enqueue(
          pid32         pid,            /* ID of process to insert      */
          qid16         q               /* ID of queue to use           */
        )
{
        qid16   tail, prev;             /* Tail & previous node indexes */

        if (isbadqid(q) || isbadpid(pid)) {
                return SYSERR;
        }

        tail = queuetail(q);
        prev = queuetab[tail].qprev;

        queuetab[pid].qnext  = tail;    /* Insert just before tail node */
        queuetab[pid].qprev  = prev;
        queuetab[prev].qnext = pid;
        queuetab[tail].qprev = pid;
        return pid;
}

/*------------------------------------------------------------------------
 *  dequeue  -  Remove and return the first process on a list
 *------------------------------------------------------------------------
 */
pid32   dequeue(
          qid16         q               /* ID queue to use              */
        )
{
        pid32   pid;                    /* ID of process removed        */

        if (isbadqid(q)) {
                return SYSERR;
        } else if (isempty(q)) {
                return EMPTY;
        }
```

```
        pid = getfirst(q);
        queuetab[pid].qprev = EMPTY;
        queuetab[pid].qnext = EMPTY;
        return pid;
}
```

Function *enqueue* calls *isbadpid* to check whether its argument is a valid process ID. The next chapter shows that *isbadpid* consists of an inline function that checks whether the ID is in the correct range and that a process with that ID exists.

File *queue.c* includes *xinu.h*, which includes the complete set of Xinu include files:

```
/* xinu.h - include all system header files */

#include <kernel.h>
#include <conf.h>
#include <process.h>
#include <queue.h>
#include <resched.h>
#include <mark.h>
#include <semaphore.h>
#include <memory.h>
#include <bufpool.h>
#include <clock.h>
#include <ports.h>
#include <uart.h>
#include <tty.h>
#include <device.h>
#include <interrupt.h>
#include <file.h>
#include <rfilesys.h>
#include <rdisksys.h>
#include <lfilesys.h>
#include <ether.h>
#include <net.h>
#include <ip.h>
#include <arp.h>
#include <udp.h>
#include <dhcp.h>
#include <icmp.h>
#include <tftp.h>
#include <name.h>
#include <shell.h>
```

```
#include <date.h>
#include <prototypes.h>
#include <delay.h>
#include <pci.h>
#include <quark_eth.h>
#include <quark_pdat.h>
#include <quark_irq.h>
#include <multiboot.h>
#include <stdio.h>
#include <string.h>
```

Combining the set of header files into a single include file helps programmers because it ensures that all pertinent definitions are available and guarantees that the set of include files are processed in a valid sequence. Later chapters consider the contents of each include file.

4.8 Manipulation Of Priority Queues

A process manager often needs to select from a set of processes the process that has highest priority. Consequently, the linked list routines must be able to maintain lists of processes that each have an associated *priority*. In our example system, a priority is an integer value assigned to the process. In general, the task of examining a process with highest priority is performed frequently compared with the tasks of inserting and deleting processes. Thus, a data structure used to manage lists of processes should be designed to make finding the highest priority process efficient compared to insertion or deletion.

A variety of data structures has been devised to store a set of items that can be selected in priority order. Any such data structure is known generically as a *priority queue*. Our example system uses a linear list to store a priority queue where the priority of a process serves as a key in the list. Because the list is ordered in descending order by key, the highest priority process can always be found at the head of the list. Thus, finding the highest priority process takes constant time. Insertion is more expensive because the list must be searched to determine the location at which a new item should be inserted.

In a small embedded system where one only expects two or three processes to be on a given priority queue at any time, a linear list suffices. For a large system where many items appear in a given priority queue or where the number of insertions is high compared to the number of times items are extracted, a linear list can be inefficient. An exercise considers the point further.

Deletion from an ordered list is trivial: the first node is removed from the list. When an item is inserted, list order must be maintained. *Insert*, which is shown below, inserts a process on a list ordered by priority. The function takes three arguments: the ID of a process to be inserted, the ID of a queue on which to insert the process, and an

integer priority for the process. *Insert* uses the *qkey* field in *queuetab* to store the process's priority. To find the correct location in the list, *insert* searches for an existing element with a key less than the key of the element being inserted. During the search, integer *curr* moves along the list. The loop must eventually terminate because the key of the tail element contains a value less than the smallest valid key. Once the correct location has been found, *insert* changes the necessary pointers to link the new node into the list.

```
/* insert.c - insert */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  insert  -  Insert a process into a queue in descending key order
 *------------------------------------------------------------------------
 */
status  insert(
          pid32         pid,            /* ID of process to insert      */
          qid16         q,              /* ID of queue to use           */
          int32         key             /* Key for the inserted process */
        )
{
        int16   curr;                   /* Runs through items in a queue*/
        int16   prev;                   /* Holds previous node index    */

        if (isbadqid(q) || isbadpid(pid)) {
                return SYSERR;
        }

        curr = firstid(q);
        while (queuetab[curr].qkey >= key) {
                curr = queuetab[curr].qnext;
        }

        /* Insert process between curr node and previous node */

        prev = queuetab[curr].qprev;    /* Get index of previous node   */
        queuetab[pid].qnext = curr;
        queuetab[pid].qprev = prev;
        queuetab[pid].qkey = key;
        queuetab[prev].qnext = pid;
        queuetab[curr].qprev = pid;
        return OK;
}
```

4.9 List Initialization

The functions described above all assume that although it may be empty, a given queue has been initialized. We now consider the code that creates an empty list. It is appropriate that the code to create an empty list occurs at the end of the chapter because it brings up an important point about the design process:

Initialization is the final step in design.

It may seem strange to defer initialization because a designer cannot postpone thinking about initialization altogether. However, the general paradigm can be stated as follows: first, design the data structures needed when the system is running, and then figure out how to initialize the data structures. Partitioning the "steady state" aspect of the system from the "transient state" helps focus the designer's attention on the most important aspect, and avoids the temptation of sacrificing good design for easy initialization.

Initialization of entries in the *queuetab* structure is performed on demand as entries are needed. A running process calls function *newqueue* to create a new list. The system maintains a global pointer to the next unallocated element of *queuetab*, which makes it easy to allocate the list.

In theory, the head and tail of a list could be allocated from any of the unused entries in *queuetab*. In practice, however, choosing arbitrary locations would require a caller to store two items: the indices of the head and tail. To optimize storage, we make the rule that:

The head and tail nodes for a list, X, are allocated from successive locations in the queuetab *array, and list X is identified by the index of the head.*

In the code, *newqueue* allocates a pair of adjacent positions in the *queuetab* array to use as head and tail nodes, and initializes the list to empty by pointing the successor of the head to the tail and the predecessor of the tail to the head. *Newqueue* assigns the value *EMPTY* to unused pointers (i.e., the successor of the tail and the predecessor of the head). When it initializes a list, *newqueue* also sets the key fields in the head and the tail to the maximum and minimum integer values, respectively, with the assumption that neither value will be used as a key. Only one allocation function is needed because a list can be used to implement a FIFO queue or a priority queue.

Once it finishes with initialization, *newqueue* returns the index of the list head to its caller. The caller only needs to store one value because the ID of the tail can be computed by adding 1 to the ID of the head.

```
/* newqueue.c - newqueue */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  newqueue  -  Allocate and initialize a queue in the global queue table
 *------------------------------------------------------------------------
 */
qid16   newqueue(void)
{
        static qid16    nextqid=NPROC;  /* Next list in queuetab to use */
        qid16           q;              /* ID of allocated queue        */

        q = nextqid;
        if (q > NQENT) {                /* Check for table overflow     */
                return SYSERR;
        }

        nextqid += 2;                   /* Increment index for next call*/

        /* Initialize head and tail nodes to form an empty queue */

        queuetab[queuehead(q)].qnext = queuetail(q);
        queuetab[queuehead(q)].qprev = EMPTY;
        queuetab[queuehead(q)].qkey  = MAXKEY;
        queuetab[queuetail(q)].qnext = EMPTY;
        queuetab[queuetail(q)].qprev = queuehead(q);
        queuetab[queuetail(q)].qkey  = MINKEY;
        return q;
}
```

4.10 Perspective

Using a single data structure for process lists makes it possible to create general-purpose linked list manipulation functions, which reduce the size of the code by avoiding duplication. Using an implicit data structure with relative pointers reduces the memory used. For small embedded systems, compacting code and data is necessary. What about systems that have plenty of memory? Interestingly, a general principle applies: unless care is taken, successive generations of software expand to fill whatever memory is available. Thus, thinking carefully about a design is always important: there are never sufficient resources to justify wasteful inefficiency.

4.11 Summary

The chapter describes a set of linked-list functions used to store processes. In our example system, linked lists of processes are kept in a single, uniform data structure, the *queuetab* array. Functions that manipulate lists of processes can produce FIFO queues or priority queues. All lists have the same form: they are doubly linked, each has both a head and tail, and each node has an integer key field. Keys are used when the list is a priority queue; keys are ignored if the list is a FIFO queue.

The implementation uses a single array for all list elements — an entry in the array either corresponds to a process or to a list head or tail. To reduce the size required, the Xinu implementation uses relative pointers and an implicit data structure. In addition, the head and tail of a list are allocated sequentially, allowing a single relative pointer to identify both the head and tail.

EXERCISES

4.1 In what sense is the queue structure described here an *implicit* data structure?

4.2 If priority values range from –8 to +8, how many bits are required to store each key in *queuetab*?

4.3 Create a separate set of functions that allows one to create a singly-linked list, and insert items in either FIFO or priority order. By how much does the second set of routines increase memory usage? Does having a separate set of routines decrease processor usage? Explain.

4.4 Does *insert* work correctly for all possible key values? If not, for which key(s) does it fail?

4.5 Implement functions to manipulate lists using pointers instead of indices into an array of structures. What is the difference in memory space and processor time?

4.6 Compare the complexity of functions like *isempty* implemented with pointers and with array indexing.

4.7 Larger systems sometimes use a data structure known as a *heap* to contain a priority queue. What is a heap? Will its use be more or less expensive than an ordered, doubly linked list when the list size is between 1 and 3?

4.8 Functions *getfirst*, *getlast*, and *getitem* do not check whether their argument is a valid queue ID. Modify the code to insert the appropriate checks.

4.9 The code generated to convert a subscript into a memory address may use multiplication. Try padding the size of a *qentry* to a power of two bytes, and examine the resulting code to see if the compiler uses a shift instead of multiplication.

4.10 In the previous exercise, measure a series of insertions and deletions to determine the difference in speed between the padded and unpadded versions of the data structure.

4.11 If a structure contains data items that are not multiples of four bytes on an architecture with strict word alignment (e.g., some RISC architectures), the code a compiler generates to access a structure member may include masking and shifting. Try altering the fields of qentry so that members are aligned on machine word boundaries, and examine the impact on the size of the queue table and the resulting code for accessing members.

Chapter Contents

5.1 Introduction, 75
5.2 The Process Table, 76
5.3 Process States, 79
5.4 Ready And Current States, 80
5.5 A Scheduling Policy, 80
5.6 Implementation Of Scheduling, 81
5.7 Deferred Rescheduling, 85
5.8 Implementation Of Context Switching, 85
5.9 State Saved In Memory, 86
5.10 Context Switch Operation, 87
5.11 An Address At Which To Restart A Process, 91
5.12 Concurrent Execution And A Null Process, 92
5.13 Making A Process Ready And The Scheduling Invariant, 93
5.14 Other Process Scheduling Algorithms, 94
5.15 Perspective, 95
5.16 Summary, 95

5

Scheduling And Context Switching

What is called a sincere work is one that is endowed with enough strength to give reality to an illusion.

— Max Jacob

5.1 Introduction

An operating system achieves the illusion of concurrent execution by rapidly switching a processor among several computations. Because the speed of the computation is extremely fast compared to that of a human, the effect is impressive — multiple activities appear to proceed simultaneously.

Context switching, which lies at the heart of the juggling act, consists of stopping the current process, saving enough information so it may be restarted later, and starting another process. What makes such a change difficult is that the processor cannot be stopped during a context switch — the processor must continue to execute the code that switches to a new process.

This chapter describes the basic context switching mechanism, showing how an operating system saves the state information from one process, chooses another process to run from among those that are ready, and relinquishes control to the new process. The chapter describes the data structure that holds information about processes that are not currently executing, and explains how the context switch uses the data structure. For the present, we will ignore the questions of when and why processes choose to switch context. Later chapters answer the questions, showing how higher levels of the operating system use the context switch described here.

5.2 The Process Table

An operating system keeps all information about processes in a data structure known as a *process table*. A process table contains one entry for each process that currently exists. We will see that a new entry must be allocated each time a process is created and an entry is removed when a process terminates. Because exactly one process is executing at any time, exactly one of the entries in a process table corresponds to an active process — the saved state information in the process table is out of date for the executing process. Each of the other process table entries contains information about a process that has been stopped temporarily. To switch context, the operating system saves information about the currently running process in its process table entry, and restores information from the process table entry corresponding to the process it is about to execute.

Exactly what information must be saved in the process table? The system must save any values that will be destroyed when the new process runs. Consider the stack. Because each process has its own separate stack memory, a copy of the entire stack need not be saved. However, when it executes, a process will change the hardware stack pointer register. Therefore, the contents of the stack pointer register must be saved when a process temporarily stops executing, and must be restored when the process resumes execution. Similarly, copies of other general-purpose registers must be saved and restored. In addition to values from the hardware, an operating system keeps meta-information in the process table. We will see how operating system functions use the meta-information for resource accounting, error prevention, and other administrative tasks. For example, the process table on a multi-user system stores the user ID of the user who owns the process. Similarly, if the operating system places policy limits on the memory that a process can allocate, the limit might be placed in the process table. The details of items in the process table will become clear in later chapters as we examine operating system functions that operate on processes.

Like most operating systems, our example system places a fixed upper bound on the maximum number of active processes. In the code, constant *NPROC* specifies the bound; the process table, *proctab*, consists of an array with *NPROC* entries. Each entry in *proctab* consists of a *procent* structure that defines the information kept for a process. Figure 5.1 lists key items found in a process table entry.

Throughout the operating system, each process is identified by an integer process ID. The following rule gives the relationship between process IDs and the process table:

> *A process is referenced by its process ID, which is the index of the proctab entry that contains the process's saved state information.*

Field	Purpose
prstate	The current status of the process (e.g., whether the process is currently executing or waiting)
prprio	The scheduling priority of the process
prstkptr	The saved value of the process's stack pointer when the process is not executing
prstkbase	The address of the highest memory location in the memory region used as the process's stack
prstklen	A limit on the maximum size that the process's stack can grow
prname	A name assigned to the process that humans use to identify the process's purpose

Figure 5.1 Key items found in the Xinu process table.

As an example of using a process ID, consider how the code finds information about a process. The state information for a process with ID 3 can be found in proctab[3], and the state information for a process with ID 5 can be found in proctab[5]. Using the array index as an ID makes locating information efficient. Of course, using an array index has a disadvantage: once a process terminates, the process ID will be reused the next time the array entry is used. We will see that the implementation tries to maximize the time between reuse of an ID. An alternative, used in some operating systems, allocates process IDs from a much larger set. Doing so has the advantage of not reusing IDs as frequently, but the disadvantage of requiring a mapping between an ID and a process table slot. One of the exercises suggests an alternative that combines the advantages of both approaches.

Each entry in *proctab* is defined to be a struct of type *procent*. The declaration of struct *procent* can be found in file *process.h* along with other declarations related to processes. Some fields in the process table contain information that the operating system needs to manage the process (e.g., the information needed to free the process's stack memory when the process completes). Other fields are used only for debugging. For example, field *prname* contains a character string that identifies the process; the field is not used except when a human tries to debug a problem or understand the current set of processes and to which computation each corresponds.

```
/* process.h - isbadpid */

/* Maximum number of processes in the system */

#ifndef NPROC
#define NPROC           8
#endif

/* Process state constants */

#define PR_FREE         0       /* Process table entry is unused        */
#define PR_CURR         1       /* Process is currently running         */
#define PR_READY        2       /* Process is on ready queue            */
#define PR_RECV         3       /* Process waiting for message          */
#define PR_SLEEP        4       /* Process is sleeping                  */
#define PR_SUSP         5       /* Process is suspended                 */
#define PR_WAIT         6       /* Process is on semaphore queue        */
#define PR_RECTIM       7       /* Process is receiving with timeout    */

/* Miscellaneous process definitions */

#define PNMLEN          16      /* Length of process "name"             */
#define NULLPROC        0       /* ID of the null process               */

/* Process initialization constants */

#define INITSTK         65536   /* Initial process stack size           */
#define INITPRIO        20      /* Initial process priority             */
#define INITRET         userret /* Address to which process returns     */

/* Inline code to check process ID (assumes interrupts are disabled)    */

#define isbadpid(x)     ( ((pid32)(x) < 0) || \
                          ((pid32)(x) >= NPROC) || \
                          (proctab[(x)].prstate == PR_FREE))

/* Number of device descriptors a process can have open */

#define NDESC           5       /* must be odd to make procent 4N bytes */

/* Definition of the process table (multiple of 32 bits) */

struct procent {                /* Entry in the process table           */
        uint16  prstate;        /* Process state: PR_CURR, etc.         */
        pri16   prprio;         /* Process priority                     */
```

```
        char    *prstkptr;      /* Saved stack pointer                  */
        char    *prstkbase;     /* Base of run time stack               */
        uint32  prstklen;       /* Stack length in bytes                */
        char    prname[PNMLEN]; /* Process name                         */
        sid32   prsem;          /* Semaphore on which process waits     */
        pid32   prparent;       /* ID of the creating process           */
        umsg32  prmsg;          /* Message sent to this process         */
        bool8   prhasmsg;       /* Nonzero iff msg is valid             */
        int16   prdesc[NDESC];  /* Device descriptors for process       */
};

/* Marker for the top of a process stack (used to help detect overflow) */
#define STACKMAGIC      0x0A0AAAA9

extern  struct  procent proctab[];
extern  int32   prcount;        /* Currently active processes           */
extern  pid32   currpid;        /* Currently executing process          */
```

5.3 Process States

To record exactly what each process is doing and to validate operations performed on the process, each process is assigned a *state*. An operating system designer defines the set of possible states as the design proceeds. Because many of the system functions that operate on processes use the state to determine whether an operation is valid, the set of states must be completely defined before the system can be implemented.

Xinu uses field *prstate* in the process table to record state information for each process. The system defines seven valid states and a symbolic constant for each. The system also defines an additional constant that is assigned when a given table entry is unused (i.e., no process has been created to use that particular entry). File *process.h* contains the definitions; Figure 5.2 lists the symbolic state constants and the meaning of each.

Because it runs as an embedded system, Xinu keeps the code and data for all processes in memory at all times. In larger operating systems, where a process executes an application program, the system can move a process to secondary storage when the process is not currently executing. Thus, in those systems, the process state also determines whether the process must reside in memory or can be moved to disk temporarily.

Constant	Meaning
PR_FREE	The entry in the process table is unused (not really a process state)
PR_CURR	The process is currently executing
PR_READY	The process is ready to execute
PR_RECV	The process is waiting for a message
PR_SLEEP	The process is waiting for a timer
PR_SUSP	The process is suspended
PR_WAIT	The process is waiting on a semaphore
PR_RECTIM	The process is waiting for a timer or a message, whichever occurs first

Figure 5.2 The seven symbolic constants that can be assigned to the state of a process.

5.4 Ready And Current States

Later chapters will explain each process state in detail, and show how and why system functions change a process's state. The remainder of this chapter focuses on the current and ready states.

Almost every operating system includes *ready* and *current* process states. A process is classified *ready* if the process is ready (i.e., eligible) to execute, but is not currently executing; at any time, the single process executing is classified as *current*.†

5.5 A Scheduling Policy

Switching from the currently executing process to another process consists of two steps: selecting a process from among those that are eligible to use the processor, and giving control of the processor to the selected process. Software that implements the policy for selecting a process is called a *scheduler*. In Xinu, function *resched* makes the selection according to the following well-known scheduling policy:

> *At any time, the highest priority eligible process is executing. Among processes with equal priority scheduling is round-robin.*

†Recall that our initial discussions are focused on a single-core processor.

Two aspects of the policy deserve attention:

- The currently executing process is included in the set of eligible processes. Thus, if process *p* is currently executing and has a higher priority than any of the other processes, process *p* will continue to execute.
- The term *round-robin* refers to a situation in which a set of *k* processes all have the same priority and the priority of processes in the set is higher than the priority of other processes. The round-robin policy means that members of the set will receive service one after another so all members of the set have an opportunity to execute before any member has a second opportunity.

5.6 Implementation Of Scheduling

The key to understanding a scheduler lies in knowing that a scheduler is merely a function. That is, the operating system scheduler is not an active agent that picks up the processor from one process and moves it to another. Instead, a running process invokes the scheduler:†

> *A scheduler consists of a function that a running process calls to willingly give up the processor.*

Recall that a process priority consists of a positive integer, and the priority for a given process is stored in the *prprio* field of the process's entry in the process table. A user assigns a priority to each process to control how the process will be selected for execution. A variety of complex scheduling policies have been proposed and measured, including schedulers that adjust the priority of processes dynamically, based on the observed behavior of each process. For most embedded systems, process priorities remain relatively static (typically, the priority does not change after a process has been created).

To make the selection of a new process fast, our example system stores all ready processes on a list known as a *ready list*. Processes on a ready list are stored in descending order by priority. Thus, a highest priority process is immediately accessible at the head of the list.

In the example code, the ready list is stored in the *queuetab* array described in Chapter 4, and the scheduler uses functions from Chapter 4 to update and access the list. That is, the key in each element on the ready list consists of the priority for the process to which the element corresponds. Global variable *readylist* contains the queue ID of the ready list.

Should an operating system keep the current process on the ready list? The answer depends on details of the implementation. Either possibility is feasible provided an entire system follows one approach or the other. Xinu implements the following policy:

---

†Later chapters explain how and why a process invokes the scheduler.

The current process does not appear on the ready list; to provide fast access to the current process, its ID is stored in global integer variable currpid.

Consider what happens when the processor switches from one process to another. The currently executing process relinquishes the processor. Often, the process that was executing remains eligible to use the processor again. In such situations, the scheduler must change the state of the current process to *PR_READY* and insert the process onto the ready list, ensuring that it will be considered for service again later. In other cases, however, the current process does not remain ready to execute, which means the process should not be placed on the ready list.

How does the scheduler decide whether to move the current process onto the ready list? In Xinu, the scheduler does not receive an explicit argument that specifies the disposition of the current process. Instead, the system functions use an implicit argument: if the current process should not remain ready, before calling *resched*, the current process's *prstate* field must be set to the desired next state. Whenever it prepares to switch to a new process, *resched* checks the *prstate* field of the current process. If the state still specifies *PR_CURR*, *resched* assumes the process should remain ready, and moves the process to the ready list. Otherwise, *resched* assumes the next state has already been chosen. The next chapter shows an example.

In addition to moving the current process to the ready list, *resched* completes every detail of scheduling and context switching except saving and restoring machine registers (which cannot be done directly in a high-level language like C). *Resched* selects a new process to run, updates the process table entry for the new process, removes the new process from the ready list, marks it current, and updates *currpid*. It also resets the preemption counter, something we will consider later. Finally, *resched* calls function *ctxsw* to save the hardware registers of the current process and restore the registers for the new process. The code can be found in file *resched.c*:

```
/* resched.c - resched, resched_cntl */

#include <xinu.h>

struct  defer   Defer;

/*------------------------------------------------------------------------
 *  resched  -  Reschedule processor to highest priority eligible process
 *------------------------------------------------------------------------
 */
void    resched(void)           /* Assumes interrupts are disabled      */
{
        struct procent *ptold;  /* Ptr to table entry for old process   */
        struct procent *ptnew;  /* Ptr to table entry for new process   */
```

```
        /* If rescheduling is deferred, record attempt and return */

        if (Defer.ndefers > 0) {
                Defer.attempt = TRUE;
                return;
        }

        /* Point to process table entry for the current (old) process */

        ptold = &proctab[currpid];

        if (ptold->prstate == PR_CURR) {  /* Process remains eligible */
                if (ptold->prprio > firstkey(readylist)) {
                        return;
                }

                /* Old process will no longer remain current */

                ptold->prstate = PR_READY;
                insert(currpid, readylist, ptold->prprio);
        }

        /* Force context switch to highest priority ready process */

        currpid = dequeue(readylist);
        ptnew = &proctab[currpid];
        ptnew->prstate = PR_CURR;
        preempt = QUANTUM;              /* Reset time slice for process */
        ctxsw(&ptold->prstkptr, &ptnew->prstkptr);

        /* Old process returns here when resumed */

        return;
}

/*------------------------------------------------------------------------
 *  resched_cntl  -  Control whether rescheduling is deferred or allowed
 *------------------------------------------------------------------------
 */
status  resched_cntl(           /* Assumes interrupts are disabled      */
          int32 defer           /* Either DEFER_START or DEFER_STOP     */
        )
{
        switch (defer) {
```

```
            case DEFER_START:   /* Handle a deferral request */

                if (Defer.ndefers++ == 0) {
                        Defer.attempt = FALSE;
                }
                return OK;

            case DEFER_STOP:    /* Handle end of deferral */
                if (Defer.ndefers <= 0) {
                        return SYSERR;
                }
                if ( (--Defer.ndefers == 0) && Defer.attempt ) {
                        resched();
                }
                return OK;

            default:
                return SYSERR;
        }
}
```

Resched begins by checking global variable *Defer.ndefers* to see whether rescheduling is deferred. If so, *resched* sets global variable *Defer.attempt* to indicate that an attempt was made during the deferral period and returns to the caller. As the next section explains, deferred rescheduling is provided for situations where the operating system must make multiple processes eligible to use the processor before allowing any to run. In particular, some I/O hardware delivers multiple data items on a single interrupt. If multiple processes are each waiting to read a data item, the operating system must handle all of them at the same time. For now, it is sufficient to understand that rescheduling can be deferred temporarily.

Once it passes the test for deferral, *resched* examines the implicit parameter described above: the state of the current process. If the state variable contains *PR_CURR* and the current process's priority is the highest in the system, *resched* returns and the current process remains running. If the state specifies that the current process should remain eligible to use the processor but the current process does not have the highest priority, *resched* adds the current process to the ready list. *Resched* then removes the process at the head of the ready list (the highest priority process), and performs a context switch.

It may be difficult to envision how context switching occurs because each concurrent process has its own instruction pointer. To see how concurrency operates, suppose that process P_1 is running and calls *resched*. If *resched* chooses to switch to process P_2, process P_1 will be stopped in the call to *ctxsw*. Once process P_2 starts to run, it can execute arbitrary code. At a later time when *resched* switches back to process P_1, execution will resume where it left off — in the call to *ctxsw*. The location at which P_1

is executing does not change just because P_2 used the processor. When process P_1 runs, the call to *ctxsw* will return to *resched*. A later section explains exactly how a context switch handles the details.

5.7 Deferred Rescheduling

Although our scheduling policy requires the highest priority eligible process to execute, *resched* permits scheduling to be deferred temporarily. The motivation arises because some operating system functions move multiple processes onto the ready list at the same time. For example, consider a timer. If two or more processes choose delays that expire at the exact same time, the operating system must move all of them to the ready state. The important idea is that one or more of the processes that become ready may have a higher priority than the process that is currently executing. However, rescheduling in the midst of such a move can result in incomplete and incorrect operation. In particular, rescheduling after making one process ready may cause the process to execute, even if other processes have higher priority. We will see an example in Chapter 7; for now, it is sufficient to understand that the situation can arise.

The solution for multiple processes consists of temporarily suspending the scheduling policy. Function *resched_cntl* provides the mechanism. At any time, a process can make the call:

```
resched_cntl(DEFER_START)
```

to defer rescheduling, and then call:

```
resched_cntl(DEFER_STOP)
```

to end a deferral period and continue normal operation.

To permit nested function calls that each request deferral, our implementation uses a global counter, *Defer.ndefers*, that is initialized to zero. When a deferral is requested, the code increments *Defer.ndefers*. Later, when a function ends its deferral period, *Defer.ndefers* is decremented. As long as the count remains positive, resched only records that a call was made, but returns to its caller without switching context. When *Defer.ndefers* reaches zero, *resched_cntl* examines *Defer.attempt* to see if *resched* was called during the deferral period. If so, *resched_cntl* invokes *resched* before returning to its caller.

5.8 Implementation Of Context Switching

Because registers and hardware status cannot be manipulated directly with a high-level language, *resched* calls an assembly language function, *ctxsw*, to switch context from one process to another. The code for *ctxsw* is, of course, machine dependent. The last step consists of resetting the program counter (i.e., jumping to the location in the

new process at which execution should resume). In Xinu, the text segment for the new process will be present in memory because Xinu keeps all parts of the program resident. The point is that the operating system must load all other state variables for the new process before jumping to the new process. Some architectures contain a pair of instructions that are used in context switching: one stores processor state information in successive memory locations and the other loads processor state from successive memory locations. On such architectures, context switching code executes a single instruction to save the processor state on the current process's stack and another instruction to load the processor state from the new process's stack. Of course, each instruction takes many cycles. RISC architectures often implement *ctxsw* with a sequence of instructions that each save one of the registers and a sequence of instructions that each load a register.

5.9 State Saved In Memory

To understand how *ctxsw* saves processor states, imagine that we can look at the memory of a system that has three active processes: two of which are ready and one of which is currently executing. Each process has a private stack. The process that is currently executing is using its stack. That is, the hardware stack pointer currently points to the top of the current process's stack.† When it calls a function, the executing process pushes copies of registers on the stack and allocates space on the stack for local variables and temporary storage needed by the called function. When it returns from a function, the saved items are popped off the stack.

Our context switching function is designed to store a copy of the machine state for a process on the process's stack. That is, just before switching context, the code stores all pertinent information about the process on the process's stack. Now imagine freezing the system at a point in time, and think about the two processes that are not currently executing. When they were running, the two processes each pushed state information on their stack because each performed a context switch as the last step before turning the processor over to another process. Thus, if we look in memory, each of the processes will have saved state information on the top of their stack. Figure 5.3 illustrates the configuration.

†Recall that stacks grow downward in memory, so the top of a stack is the lowest memory address that has been used in a process's stack area.

Figure 5.3 Illustration of stacks in memory when processes (a) and (b) are on the ready list, and process (c) is currently executing.

5.10 Context Switch Operation

Our example *ctxsw* function takes two arguments: a pointer to the process table entry for the current process and a pointer to the process table entry for the new process.

- Execute instructions that push the contents of the processor register on the stack of the process that is running when *ctxsw* is called (i.e., the *old* process).
- Save the stack pointer in the process table entry for the current process, and load the stack pointer for the "new" process.
- Execute instructions that reload the processor registers from values previously saved on the new process's stack.
- Return to the function in the new process that called *ctxsw*.

Because context switching involves the direct manipulation of processor registers, the code is written in assembly language. In addition to saving copies of the general-purpose registers, most processors require a context switch to save internal hardware re-

gisters, such as the status register (i.e., the register that records whether the last arithmetic result was positive, negative, or zero).

The second step, saving and restoring stack pointers, is handled by passing two arguments to *ctxsw*. The first argument gives the address of the location in the process table where the current process's stack pointer should be stored. The second argument gives the address of the location in the process table where the stack pointer for the new process has previously been saved. Thus, when saving the stack pointer for the current process, the context switch merely needs to dereference the first argument. Similarly, to pick up the stack pointer for the new process, the context switch merely needs to dereference the second argument. We will see that on an Intel processor, one of the general purpose registers must be saved because dereferencing requires the use of one register. On the ARM architecture, the calling sequence requires the caller to save registers *r0* through *r3* (which are used to pass arguments). Because the context switch function only has two arguments, only registers *r0* and *r1* contain argument values. Therefore, the ARM context switch code can use either of registers *r2* or *r3*.

After the second step, the hardware stack pointer points to the new process's stack. *Ctxsw* extracts the set of values that was saved on the stack for the process, and loads the values into processor registers. The following sections show the code for the Intel and ARM platforms.

5.10.1 Galileo (Intel)

On an Intel platform, such as the Galileo, the context switch begins by saving values for the old process on the current stack (i.e., the old process's stack). It first pushes the *EBX* register (so *EBX* can be used to access the arguments). It then pushes the processor flags and all general-purpose registers. The *flags* contain the current processor status. The code then saves the old process's stack pointer in the location given by the first argument, and picks up the new process's stack pointer from the location given by the second argument. Once the stack pointer has been switched, *ctxsw* restores values that were previously saved with the new process: the general-purpose registers, the flags, and the saved value of *EBX*. Finally, *ctxsw* returns. Note that when the return occurs, the new process will be running. An Intel processor includes a single machine instruction to push copies of all the registers on the stack (*pushal*), and a single machine instruction to restore all the registers from the saved values (*popal*). File *ctxsw.S* contains the code.

```
/* ctxsw.S - ctxsw (for x86) */

                .text
                .globl  ctxsw

/*------------------------------------------------------------------------
 * ctxsw -  X86 context switch; the call is ctxsw(&old_sp, &new_sp)
 *------------------------------------------------------------------------
 */
```

```
ctxsw:
                pushl   %ebp            /* Push ebp onto stack          */
                movl    %esp,%ebp       /* Record current SP in ebp     */
                pushfl                  /* Push flags onto the stack    */
                pushal                  /* Push general regs. on stack  */

                /* Save old segment registers here, if multiple allowed */

                movl    8(%ebp),%eax    /* Get mem location in which to */
                                        /*   save the old process's SP  */
                movl    %esp,(%eax)     /* Save old process's SP        */
                movl    12(%ebp),%eax   /* Get location from which to   */
                                        /*   restore new process's SP   */

                /* The next instruction switches from the old process's */
                /*   stack to the new process's stack.                  */

                movl    (%eax),%esp     /* Pop up new process's SP      */

                /* Restore new seg. registers here, if multiple allowed */

                popal                   /* Restore general registers    */
                movl    4(%esp),%ebp    /* Pick up ebp before restoring */
                                        /*   interrupts                 */
                popfl                   /* Restore interrupt mask       */
                add     $4,%esp         /* Skip saved value of ebp      */
                ret                     /* Return to new process        */
```

5.10.2 BeagleBone Black (ARM)

The context switch for an ARM platform, such as the BeagleBone Black, follows almost the same approach as on the Intel platform. On an ARM processor, the co-processor stores the internal hardware status register. Therefore, to save a copy of the status, the context switch must obtain the value from the co-processor. Instruction *mrs* moves the status from the co-processor status register to a specified general-purpose register. As described above, the example code uses register *r3* because the calling sequence allows a called function to change *r3*. Once a copy of the status has been obtained, the code saves a copy of registers *r3* through *r12* and register *lr* on the current process's stack. The code then saves the old process's stack pointer in the location given by the first argument, picks up the new process's stack pointer from the location given by the second argument, restores registers *r3* through *r12* and lr from the stack, restores the status register from *r3*, and returns.

Because it uses a RISC approach, an ARM processor does not have a single machine instruction that can save multiple registers, nor does the processor have a single machine instruction that can restore multiple registers. Instead, a given instruction can only save (i.e., push) or restore (i.e., pop) a single register. Thus, one might expect the context switch code to start with a series of statements that each push one register:

```
push    r3
push    r4
push    r5
push    r6
push    r7
push    r8
push    r9
push    r10
push    r11
push    r12
push    lr
```

and to end with a series of statements that each pop one register:

```
pop     lr
pop     r12
pop     r11
pop     r10
pop     r9
pop     r8
pop     r7
pop     r6
pop     r5
pop     r4
pop     r3
```

Interestingly, the code does not contain such sequences. Instead, the code uses assembler *directives* that the assembler interprets and uses to generate multiple instructions. For example, the *push* directive

```
push    {r3–r12, lr}
```

instructs the assembler to generate a sequence of instructions that each *push* one of the registers listed. Similarly, the *pop* directive:

```
pop     {r3–r12, lr}
```

causes the assembler to generate a sequence of *pop* instructions. The assembler generates the *pop* sequence in the opposite order than the *push* sequence, which gives the instruction sequence illustrated above. File *ctxsw.S* contains the code.

```
/* ctxsw.S - ctxsw (for ARM) */

        .text
        .globl  ctxsw

/*------------------------------------------------------------------------
 * ctxsw -  ARM context switch; the call is ctxsw(&old_sp, &new_sp)
 *------------------------------------------------------------------------
 */

ctxsw:
        push    {r0-r11, lr}            /* Push regs 0 - 11 and lr      */
        push    {lr}                    /* Push return address          */
        mrs     r2, cpsr                /* Obtain status from coprocess.*/
        push    {r2}                    /*   and push onto stack        */
        str     sp, [r0]                /* Save old process's SP        */
        ldr     sp, [r1]                /* Pick up new process's SP     */
        pop     {r0}                    /* Use status as argument and   */
        bl      restore                 /*   call restore to restore it */
        pop     {lr}                    /* Pick up the return address   */
        pop     {r0-r12}                /* Restore other registere      */
        mov     pc, r12                 /* Return to the new process    */
```

5.11 An Address At Which To Restart A Process

A potential problem arises during context switching because the processor can change registers. Thus, the code must be written carefully because once a given register has been saved, subsequent changes will be lost when the process restarts. Fortunately, the standard calling sequence helps by saving and restoring registers when the context switch function is called. The instruction pointer (i.e., program counter) represents a special dilemma, because storing the value means that when the process restarts, execution will continue at exactly the point in the code at which the instruction pointer was stored. If a copy of the instruction pointer is saved before the context switch has completed, the process will restart at a point *before* the context switch has occurred. The code in *ctxsw* reveals how the situation is resolved: the instruction pointer is not among the registers saved on the stack. Instead, we only save the address to which *ctxsw* should return.

To understand the idea, think of an executing process, *P*, that has called *resched* which has then called *ctxsw*. We assume that the only way *P* will regain the processor later will occur when some other process calls *ctxsw*. So, if we save the return address, when *P* runs again, *ctxsw* will return as if it were a normal function. Consequently:

> *When a process restarts, the process will resume execution in* resched *immediately following the call to* ctxsw.

We assume that no context switching occurs outside of *ctxsw*. That is, all processes must call *resched* to perform context switching, and *resched* calls *ctxsw*. Thus, if one were to freeze the system at an arbitrary instant and examine memory, the saved information for each ready process will have the same value for a return address — an address just after the call to *ctxsw* in *resched*. However, each process has its own stack of function calls, which means that when a given process resumes execution and returns from *resched*, the return may go to a different caller than the return in another process.

The notion of function return forms a key ingredient in keeping the system design clean. Function calls proceed downward through each level of the system, and each call returns. To enforce the design at all levels, the scheduler, *resched*, and the context switch, *ctxsw*, have each been designed to behave like any other function and return. To summarize:

> *In Xinu, each function, including the scheduler and context switch, eventually returns to its caller.*

Of course, rescheduling allows other processes to execute, and the execution may take arbitrarily long (depending on process priorities). Thus, a considerable delay may elapse between a call to *resched* and the time the call returns and the process runs again.

5.12 Concurrent Execution And A Null Process

The concurrent execution abstraction is complete and absolute. That is, an operating system views all computation as part of a process — there is no way that the processor can temporarily stop executing processes and execute a separate piece of code. The scheduler design reflects the following principle: the scheduler's only function is to switch the processor among the set of processes that are current or ready. The scheduler cannot execute any code that is not part of a process, and cannot create a new process. Figure 5.4 illustrates the possible state transitions.

Figure 5.4 Illustrations of state transitions for processes between the ready and current states.

We will see that a given process does not always remain ready to execute. For example, a process stops executing when it waits for I/O to complete or when it needs to use a shared resource that is already in use. What happens if all processes wait for I/O? *Resched* will fail because the code has been designed to assume at least one process will be eligible to execute at any time. When the currently executing process blocks, *resched* removes the first process from the ready list without verifying that the list is nonempty. If the list is empty, an error results. To summarize:

> *Because an operating system can only switch the processor from one process to another, at least one process must remain ready to execute at all times.*

To ensure that at least one process always remains ready to execute, Xinu uses a standard technique: it creates an extra process, called the *null process*, when the system boots. The null process has process ID zero and priority zero (a lower priority than any other process). The null process code, which will be shown in Chapter 22, consists of an infinite loop. Because all other processes must have a priority greater than zero, the scheduler switches to the null process only when no other process is ready to run. In essence, the operating system switches the processor to the null process when all other processes are blocked (e.g., waiting for I/O).†

5.13 Making A Process Ready And The Scheduling Invariant

Because it performs a swap by removing a process from the ready list and (possibly) moving the current process onto the ready list, *resched* manipulates the list directly. We will see that many other functions need to make a process eligible for processor service. The task occurs so frequently that we designed a function to do just that. The function is named *ready*.

Ready takes an argument that specifies a process ID, and makes the process eligible to execute. Our scheduling policy specifies that at any time, the highest priority eligible process *must* be executing. We say that each operating system function should maintain a *scheduling invariant*: a function assumes the highest priority process was executing when the function was called, and must ensure that the highest priority process is executing when the function returns. Thus, if a function changes the state of processes, the function must call *resched* to reestablish the invariant. Thus, when it places a high priority process on the ready list, *ready* calls *resched* to ensure that the policy is followed. File *ready.c* contains the code.

†Some processors include a special instruction that can be placed in a null process that stops the processor until an interrupt occurs; using the special instruction may reduce the energy that the processor consumes.

```
/* ready.c - ready */

#include <xinu.h>

qid16   readylist;                      /* Index of ready list          */

/*------------------------------------------------------------------------
 *  ready  -  Make a process eligible for CPU service
 *------------------------------------------------------------------------
 */
status  ready(
          pid32         pid             /* ID of process to make ready  */
        )
{
        register struct procent *prptr;

        if (isbadpid(pid)) {
                return SYSERR;
        }

        /* Set process state to indicate ready and add to ready list */

        prptr = &proctab[pid];
        prptr->prstate = PR_READY;
        insert(pid, readylist, prptr->prprio);
        resched();

        return OK;
}
```

5.14 Other Process Scheduling Algorithms

Process scheduling was once an important topic in operating systems, and many scheduling algorithms have been proposed as alternatives to the round-robin scheduler in Xinu. For example, one policy measures the amount of I/O a process performs and gives the processor to the process that spends the most time doing I/O. Proponents of the policy argue that because I/O devices are slower than processors, choosing a process that performs I/O will increase the total throughput of the system.

Because scheduling is confined to a few functions, it is easy to experiment with the scheduling policy in Xinu. Changing *resched* and *ready* changes the basic scheduler. Of course, if the new policy uses data that Xinu does not already gather (e.g., the amount of time spent doing I/O), other functions may need to change to record the appropriate data.

5.15 Perspective

The most interesting aspect of scheduling and context switching arises because they are embedded as part of normal computation. That is, instead of the operating system being implemented separately from the processes it controls, the operating system code is executed by the processes themselves. Thus, the system does not have an extra process that can stop the processor from executing one application and move it to another, scheduling and context switching occur as the side effect of a function call.

We will see that using processes to execute operating system code affects the design. When a programmer writes an operating system function, the programmer must accommodate execution by concurrent processes. Similarly, using processes to execute operating system code affects how the system interacts with I/O devices and how it handles interrupts.

5.16 Summary

Scheduling and context switching form a foundation for concurrent execution. Scheduling consists of choosing a process from among those that are eligible for execution. Context switching consists of stopping one process and starting a new one. To keep track of processes, the system uses a global data structure called a process table. Whenever it temporarily suspends a process, the context switch saves the processor state for the process on the process's stack and places a pointer to the stack in the process table. To restart a process, the context switch reloads the processor state information from the process's stack, and resumes execution in the process at the point the call to the context switch function returns.

To allow functions to determine when an operation is permitted, each process is assigned a *state*. A process that is using the processor is assigned the *current* state, and a process that is eligible to use the processor, but is not currently executing, is assigned the *ready* state. Because at least one process must remain eligible to execute at any time, the operating system creates an extra process at startup known as the *null process*. The null process has priority zero, and all other processes have priority greater than zero. Consequently, the null process only runs when no other process is eligible.

The chapter presents three functions that perform transitions between the *current* and *ready* states. Function *resched* performs scheduling, function *ctxsw* performs context switching, and function *ready* makes a process eligible to execute.

EXERCISES

5.1 If the operating system contains a total of N processes, how many processes can be on the ready list at a given time? Explain.

5.2 How do operating system functions know which process is executing at a given time?

5.3 Rewrite *resched* to have an explicit parameter giving the disposition of the currently executing process, and examine the assembly code generated to determine the number of instructions executed in each case.

5.4 What are the basic steps performed during a context switch?

5.5 Investigate another hardware architecture (e.g., SPARC or MIPS), and determine what information needs to be saved during a context switch.

5.6 How much memory is needed to store processor state for a MIPS processor? Which registers must be saved, and why? How does the standard calling convention for a processor affect the answer?

5.7 Suppose process *k* has been placed on the ready list. When process *k* becomes current, where will execution start?

5.8 Why is a null process needed?

5.9 Consider a modification to the code that stores processor state in the process table instead of on the process's stack (i.e., assume the process table entry contains an array that holds the contents of registers). What are the advantages of each approach?

5.10 In the previous exercise, does saving registers in the process table reduce or increase the number of instructions executed during a context switch?

5.11 Devise a scheduling policy for a dual-core processor (i.e., a processor that contains two separate processing cores that can execute in parallel).

5.12 Extend the previous exercise: show that executing *resched* on one core may require changing the process that is running on the other core. (Note: many operating systems for dual-core processors avoid the problem by specifying that all operating system functions, including scheduling, run on one of the two cores.)

5.13 Variable *Defer.attempt* records whether *resched* was called during the period when rescheduling is deferred, but it does not record whether a context switch would have occurred. Should the code be rewritten to record whether rescheduling is needed rather than whether *resched* was called? Why or why not?

Chapter Contents

6.1 Introduction, 99
6.2 Process Suspension And Resumption, 99
6.3 Self–suspension And Information Hiding, 100
6.4 The Concept Of A System Call, 101
6.5 Interrupt Control With Disable And Restore, 103
6.6 A System Call Template, 104
6.7 System Call Return Values SYSERR And OK, 105
6.8 Implementation Of Suspend, 105
6.9 Suspending The Current Process, 107
6.10 The Value Returned By Suspend, 107
6.11 Process Termination And Process Exit, 108
6.12 Process Creation, 111
6.13 Other Process Manager Functions, 115
6.14 Summary, 117

6

More Process Management

When men willingly suspend fear, science flourishes.

— Anonymous

6.1 Introduction

Chapter 5 discusses the concurrent execution abstraction and process execution. The chapter explains how an operating system stores information about processes in a table, and how each process is assigned a state. Chapter 5 also explains the concepts of scheduling and context switching. It shows how a scheduler implements a scheduling policy, and explains how a process moves between the ready and current states.

This chapter extends our study of the process management functions in an operating system. The chapter explains how a new process comes into existence, and what happens when a process exits. The chapter also examines a process state that allows a process to be suspended temporarily, and explores functions that move processes among the current, ready, and suspended states.

6.2 Process Suspension And Resumption

We will see that operating system functions sometimes need to stop a process from executing temporarily and, at a later time, resume execution. We say that a stopped process has been placed in a state of "suspended animation." For example, suspended animation can be used when a process waits for one of several restart conditions without knowing which will occur first.

The first step in implementing operating system functionality consists of defining a set of operations. In the case of suspended animation, only two conceptual operations provide all the functionality that is needed:

- *Suspend* stops a process and places the process in suspended animation (i.e., makes the process ineligible to use the processor).
- *Resume* continues execution of a previously suspended process (i.e., makes the process eligible to use the processor again).

Because it is not eligible to use the processor, a suspended process cannot remain in either the *ready* or *current* states. Thus, a new state must be invented. We call the new state *suspended*, and add the new state and associated transitions to the state diagram. Figure 6.1 shows the extended state diagram, which summarizes how *suspend* and *resume* affect the state. The resulting diagram documents the possible transitions among the *ready*, *current*, and *suspended* states.

Figure 6.1 Transitions among the current, ready, and suspended states.

6.3 Self–suspension And Information Hiding

Although each state transition in Figure 6.1 has a label that specifies a particular function, process suspension differs from scheduling in a significant way: instead of acting on the current process, *suspend* allows one process to suspend another process. More important, because a suspended process cannot resume itself, *resume* must allow an executing process to resume a previously suspended process. Thus, *suspend* and *resume* each take an argument that specifies the ID of a process on which the operation should be performed.

Can a process suspend itself? Yes. To do so, a process must obtain its process ID and then pass the ID as an argument to *suspend*. An implementation may seem obvious. Because global variable *currpid* contains the process ID of the currently executing process, a self-suspension can be achieved with:

```
suspend( currpid );
```

However, a well-designed operating system design adheres to the *principle of information hiding*: implementation details are not generally revealed. Thus, instead of permitting processes to access global variables like *currpid* directly, Xinu includes a function named *getpid* that a process can call to obtain its ID. Thus, to suspend itself, a process calls:

```
suspend( getpid( ) );
```

The present implementation of *getpid* merely returns the value of *currpid*, which may seem like unnecessary overhead. However, the advantage of information hiding becomes clear when one considers modifying the operating system. If all processes call *getpid*, a designer can change the details of where and how the current process ID is stored without changing other code.

The point is:

> *A good system design follows the principle of information hiding, which states that implementation details are not revealed unless necessary. Hiding such details makes it possible to change the implementation of a function without rewriting code that uses the function.*

6.4 The Concept Of A System Call

In theory, process resumption is straightforward. The process must be placed in the ready state and inserted in the correct position on the ready list. The *ready* function described in the previous chapter performs both tasks, so it may seem that *resume* is unnecessary. In practice, however, *resume* adds an extra layer of protection: it makes no assumptions about the caller or the correctness of the arguments. In particular, an arbitrary process can invoke *resume* at an arbitrary time with arbitrary arguments.

We used the term *system call* to distinguish a function like *resume* from internal functions like *ready*. In general, we think of the set of system calls as defining a view of the operating system from the outside — application processes invoke system calls to obtain services. In addition to adding a layer of protection, the system call interface provides another example of information hiding: application processes remain unaware of the internal implementation, and can only use the set of system calls to obtain service. We will see that the distinction between system calls and other functions appears throughout an operating system. To summarize:

> *System calls, which define operating system services for applications, protect the system from illegal use and hide information about the underlying implementation.*

To provide protection, system calls handle three aspects of computation that underlying functions do not:

- Check all arguments
- Ensure that changes leave global data structures in a consistent state
- Report success or failure to the caller

In essence, a system call cannot make any assumptions about the process that is making the call. Thus, instead of assuming that the caller has supplied correct and meaningful argument values, a system call checks each argument. More important, many system calls make changes to operating system data structures, such as the process table and lists of processes stored in the queue structure. A system call must guarantee that no other process will attempt to change data structures at the same time, or inconsistencies can result. Because it cannot make assumptions about the conditions under which it will be invoked, a system call must take steps to prevent other processes from executing concurrently while data structures are being changed. There are two aspects:

- Avoid invoking any functions that voluntarily relinquish the processor
- Disable interrupts to prevent involuntarily relinquishing the processor

To prevent voluntarily relinquishing the processor, a system call must avoid direct or indirect calls to *resched*. That is, while changes are in progress, a system call cannot invoke *resched* directly and cannot invoke any function that calls *resched*. To prevent involuntarily relinquishing the processor, a system call disables interrupts until a change is complete. In Chapter 13, we will understand the reason: hardware interrupts can result in rescheduling because some interrupt routines call *resched*.

An example system call will help clarify the two aspects. Consider the code for *resume* that is contained in file *resume.c*.

```
/* resume.c - resume */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  resume  -  Unsuspend a process, making it ready
 *------------------------------------------------------------------------
 */
pri16   resume(
          pid32         pid             /* ID of process to unsuspend   */
```

```
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  procent *prptr;         /* Ptr to process' table entry  */
        pri16   prio;                   /* Priority to return           */

        mask = disable();
        if (isbadpid(pid)) {
                restore(mask);
                return (pri16)SYSERR;
        }
        prptr = &proctab[pid];
        if (prptr->prstate != PR_SUSP) {
                restore(mask);
                return (pri16)SYSERR;
        }
        prio = prptr->prprio;           /* Record priority to return    */
        ready(pid);
        restore(mask);
        return prio;
}
```

6.5 Interrupt Control With Disable And Restore

As expected, the code in *resume* checks argument *pid* to ensure that the caller has supplied a valid process ID and the specified process is in the suspended state. Before it performs any computation, however, *resume* guarantees that no interrupts will occur (i.e., no context switching can occur until *resume* invokes an operating system function that causes a context switch). To control interrupts, *resume* uses a pair of functions:†

- Function *disable* disables interrupts and returns the previous interrupt status to its caller.
- Function *restore* reloads an interrupt status from a previously saved value.

As expected, *resume* disables interrupts immediately upon entry. *Resume* can return in two ways: either an error is detected or *resume* finishes the requested operation successfully. In either case, *resume* must call *restore* before returning to reset the interrupt status to the same value the caller was using when the call began.

Programmers who do not have experience writing code for operating systems often expect a system call to enable interrupts before returning. However, *restore* provides more generality. To see why, observe that because it restores interrupts rather than simply enabling them, *resume* works correctly whether it is called with interrupts enabled or disabled. On the one hand, if a function has interrupts disabled when it calls *resume*,

†Chapter 12 explains the details of interrupt handling.

the call will return with interrupts disabled. On the other hand, if a function has interrupts enabled when it calls *resume*, the call will return with interrupts enabled.

> *System calls must disable interrupts to prevent other processes from changing global data structures; using a disable/restore paradigm increases generality.*

6.6 A System Call Template

Another way to look at interrupt handling focuses on an invariant that a system function must maintain:

> *An operating system function always returns to its caller with the same interrupt status as when it was called.*

To ensure the invariant is maintained, operating system functions follow the general approach that Figure 6.2 illustrates.

```
syscall function_name( args )  {
        intmask mask;               /* Saved interrupt mask                  */
        mask = disable( );          /* Disable interrupts at start of function */
        if ( args are incorrect ) {
                restore(mask);      /* Restore interrupts before error return */
                return SYSERR;
        }
        ... other processing ...
        if ( an error occurs ) {
                restore(mask);      /* Restore interrupts before error return */
                return SYSERR;
        }
        ... more processing ...
        restore(mask);              /* Restore interrupts before normal return*/
        return appropriate value ;
}
```

Figure 6.2 Illustration of the general form of an operating system function.

6.7 System Call Return Values SYSERR And OK

We will see that some system calls return a value that relates to the function being performed and others merely return a status to indicate that the call was successful. *Resume* provides an example of the former: it returns the priority of the process that has been resumed. In the case of *resume*, care must be taken to record the priority before calling *ready* because the resumed process may have higher priority than the currently executing process. Thus, as soon as *ready* places the specified process on the ready list and calls *resched*, the new process may begin executing. In fact, an arbitrary delay can occur between the time *resume* calls *ready* and execution continues after the call. During the delay, an arbitrary number of other processes can execute and processes may terminate. Thus, to ensure that the returned priority reflects the resumed process's priority at the time of resumption, *resume* makes a copy in local variable *prio* before calling *ready*. *Resume* uses the local copy as the return value.

To aid in reporting status, Xinu defines two constants that are used as return values throughout the system. A function returns *SYSERR* to indicate that an error occurred during processing. That is, a system function returns *SYSERR* if the arguments are incorrect (e.g., outside the acceptable range) or the requested operation could not be completed successfully. A function such as *ready* that does not compute a specific return value uses constant *OK* to indicate that the operation was successful.

6.8 Implementation Of Suspend

As the state diagram in Figure 6.1 indicates, *suspend* can only be applied to a process that is current or ready. Suspension of a ready process is trivial: the process must be removed from the ready list and its state must be changed to suspended. No further action is required. Thus, after deleting the process from the ready list and changing the process's state to *PR_SUSP*, *suspend* can restore interrupts and return to its caller. The suspended process will remain ineligible to use the processor until it has been resumed.

Suspending the current process is almost as easy. The only subtle point is that *resched* uses an implicit argument to specify the disposition of the calling process. Recall from Chapter 5 that if a caller does not want to remain eligible to use the processor, the caller must set its state before invoking *resched*. Therefore, to suspend the current process, *suspend* must set the state of the current process to *PR-SUSP* and then call *resched*. That is, *suspend* sets the state of the current process to the desired next state.

The code for function *suspend* can be found in file *suspend.c*.

```
/* suspend.c - suspend */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  suspend  -  Suspend a process, placing it in hibernation
 *------------------------------------------------------------------------
 */
syscall suspend(
          pid32         pid             /* ID of process to suspend     */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  procent *prptr;         /* Ptr to process' table entry  */
        pri16   prio;                   /* Priority to return           */

        mask = disable();
        if (isbadpid(pid) || (pid == NULLPROC)) {
                restore(mask);
                return SYSERR;
        }

        /* Only suspend a process that is current or ready */

        prptr = &proctab[pid];
        if ((prptr->prstate != PR_CURR) && (prptr->prstate != PR_READY)) {
                restore(mask);
                return SYSERR;
        }
        if (prptr->prstate == PR_READY) {
                getitem(pid);               /* Remove a ready process   */
                                            /*   from the ready list    */
                prptr->prstate = PR_SUSP;
        } else {
                prptr->prstate = PR_SUSP;   /* Mark the current process */
                resched();                  /*   suspended and resched. */
        }
        prio = prptr->prprio;
        restore(mask);
        return prio;
}
```

Like *resume*, *suspend* is a system call, which means the function disables interrupts when it is invoked. In addition, *suspend* checks argument *pid* to verify that it is a valid

process ID. Because suspension is only valid for a process that is ready or current, the code verifies that the process is in one of the two valid states. If an error is detected, *suspend* restores interrupts and returns *SYSERR* to the caller.

6.9 Suspending The Current Process

The code for suspending the currently executing process raises two interesting points. First, the currently executing process will stop executing, at least temporarily. Thus, before suspending itself, the current process must prearrange for some other process to resume it (or it will remain suspended forever). Second, because it will be suspended, the current process must allow another process to execute. Thus, when suspending the current process, *suspend* must call *resched*. The key idea is that when a process suspends itself, the process remains executing until the call to *resched* selects another process and switches context.

Note that when a process is suspended, *resched* does not place the process on the ready list. In fact, a suspended process is not on a list of suspended processes because there is no list of suspended processes analogous to the list of ready processes. Ready processes are only kept on an ordered list to speed the search for the highest priority process during rescheduling. Because the system never searches through suspended processes looking for one to resume, the set of suspended processes need not be kept on a list. Thus, before suspending a process, a programmer must arrange a way for another process to find the ID of the suspended process so it can be resumed.

6.10 The Value Returned By Suspend

Suspend, like *resume*, returns the priority of the suspended process to its caller. In the case of a ready process, the value returned will reflect the priority the process had when *suspend* was called. (Once *suspend* disables interrupts, no other process can change priorities in the system, so the priority can be recorded at any time before *suspend* restores interrupts.) In the case of the current process, however, a question arises: should *suspend* return the priority that was in effect when *suspend* was invoked or the priority the process has after the process has been resumed (i.e., when *suspend* returns)? The two values can differ because it is possible to change the priority of a process at any time, which means the priority can change while the process is suspended. In terms of the code, the question is whether the local copy of the priority should be recorded before the call to *resched* or afterward (the version above records it afterward).

To understand one possible motivation for returning the priority at the time of resumption, consider how the priority can be used to convey information. Suppose, for example, that a process needs to suspend until one of two events occurs. A programmer can assign a unique priority value to each event (e.g., priorities 25 and 26), and arrange the calls to *resume* to set the priority accordingly. The process can then use the priority to determine why it was resumed:

```
newprio = suspend( getpid() );
if (newprio == 25) {
        ... Event 1 has occurred ...
} else {
        ... Event 2 has occurred ...
}
```

6.11 Process Termination And Process Exit

Although it freezes a process temporarily, *suspend* saves information about a process so the process can be resumed later. Another system call, *kill*, implements process termination by completely removing a process from the system. Once it has been killed, a process cannot be restarted because *kill* eradicates the entire record and frees the process table entry for reuse by a new process.

The actions taken by *kill* depend on the process state. Before writing the code, a designer needs to consider each possible process state and what it means to terminate a process in that state. We will see, for example, that a process in the ready, sleeping, or waiting states is stored on one of the linked lists in the queue structure, which means *kill* must dequeue the process. In the next chapter, we will see that if a process is waiting for a semaphore, *kill* must adjust the semaphore count. Each of the cases will become clear once we have examined the process state and the functions that control the state. For now, it is sufficient to understand the overall structure of *kill* and see how it handles processes that are current or ready. The code for *kill* appears in file *kill.c*.

Kill checks its argument, *pid*, to ensure that it corresponds to a valid process other than the null process (the null process cannot be killed because it must remain running). *Kill* then decrements *prcount*, a global variable that records the number of active user processes, and calls function *freestk* to free memory that has been allocated for the process's stack. The remaining actions depend on the process's state. For a process that is in the *ready* state, *kill* removes the process from the ready list and then frees the process table entry by assigning value *PR_FREE* to the process's state. Because it no longer appears on the ready list, the process will not be selected for rescheduling; because it has state *PR_FREE*, the entry in the process table can be reused.

Now consider what happens when *kill* needs to terminate the currently executing process. We say that the process *exits*. As before, *kill* validates its argument and decrements the count of active processes. If the current process happens to be the last user process, decrementing *prcount* makes it zero, so *kill* calls function *xdone*, which is explained below. Because *resched* uses an implicit argument to control disposition of the current process, *kill* must set the current process's state to the desired state before calling *resched*. To remove the current process from the system, *kill* sets the current process's state to *PR_FREE*, meaning that the process table slot is unused, and then calls *resched*.

```
/* kill.c - kill */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  kill  -  Kill a process and remove it from the system
 *------------------------------------------------------------------------
 */
syscall kill(
          pid32         pid             /* ID of process to kill        */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  procent *prptr;         /* Ptr to process' table entry  */
        int32   i;                      /* Index into descriptors       */

        mask = disable();
        if (isbadpid(pid) || (pid == NULLPROC)
            || ((prptr = &proctab[pid])->prstate) == PR_FREE) {
                restore(mask);
                return SYSERR;
        }

        if (--prcount <= 1) {           /* Last user process completes  */
                xdone();
        }

        send(prptr->prparent, pid);
        for (i=0; i<3; i++) {
                close(prptr->prdesc[i]);
        }
        freestk(prptr->prstkbase, prptr->prstklen);

        switch (prptr->prstate) {
        case PR_CURR:
                prptr->prstate = PR_FREE;       /* Suicide */
                resched();

        case PR_SLEEP:
        case PR_RECTIM:
                unsleep(pid);
                prptr->prstate = PR_FREE;
                break;

        case PR_WAIT:
```

```
                semtab[prptr->prsem].scount++;
                /* Fall through */

        case PR_READY:
                getitem(pid);           /* Remove from queue */
                /* Fall through */

        default:
                prptr->prstate = PR_FREE;
        }

        restore(mask);
        return OK;
}
```

When the last user process exits, *kill* calls *xdone*. In some systems, *xdone* powers down the device. In others, *xdone* restarts the device. In our example, *xdone* merely prints a message on the console, and halts the processor. The code is found in file *xdone.c*.

```
/* xdone.c - xdone */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  xdone  -  Print system completion message as last process exits
 *------------------------------------------------------------------------
 */
void    xdone(void)
{
        kprintf("\n\nAll user processes have completed.\n\n");
        halt();                         /* Halt the processor           */
}
```

Why should *kill* invoke function *xdone*? Doing so may seem unnecessary because the code is trivial and could easily be incorporated into *kill* itself. The motivation for using a function stems from a desire to separate functionality: a programmer can change the action taken when all processes exit without modifying *kill*.

A more serious question arises because *xdone* is invoked *before* the last user process has been removed from the system. To understand the problem, consider a fault-tolerant design that restarts processes in the case all processes exit. With the current implementation, one of the process table slots remains in use when *xdone* is called. The exercises consider an alternative implementation.

6.12 Process Creation

As we have seen, processes are dynamic — a process can be created at any time. The system call *create* starts a new, independent process. In essence, *create* builds an image of the process as if it had been stopped while running. Once the image has been constructed and the process has been placed on the ready list, *ctxsw* can switch to it.

A look at the code in file *create.c* explains most of the details. *Create* uses function *newpid* to search the process table for a free (i.e., unused) slot. Once a slot has been found, *create* allocates space for the new process's stack, and fills in the process table entry. *Create* calls *getstk* to allocate space for a stack (Chapter 9 discusses memory allocation).

The first argument to *create* specifies the initial function at which the process should start execution. *Create* forms a saved environment on the process's stack as if the specified function had been called. Consequently, we refer to the initial configuration as a *pseudo call*. To build a pseudo call, *create* stores initial values for the registers, including the stack pointer and a return address in the pseudo-call on the process's stack. When *ctxsw* switches to it, the new process begins executing the code for the designated function, obeying the normal calling conventions for accessing arguments and allocating local variables. In short, the initial function for a process behaves exactly as if it had been called.

What value should *create* use as a return address in the pseudo call? The value determines what action the system will take if a process returns from its initial (i.e., top-level) function. Our example system follows a well-known paradigm:

> *If a process returns from the initial (top-level) function in which its execution started, the process exits.*

To be precise, we should distinguish between a return from the function itself and a return from the initial call. To see why, observe that C permits recursive function calls. Thus, if a process begins in function *X* which recursively calls *X*, the first return merely pops one level of recursion and returns to the initial call without causing the process to exit. If the process returns again (or reaches the end of *X*) without making further calls, the process will exit.

To arrange for an exit to occur when the initial call returns, *create* assigns the address of function *userret* as the return address in the pseudo call. The code uses symbolic constant *INITRET*, which has been defined to be function name *userret*.† If during the initial call the process reaches the end of the function or explicitly invokes *return*, control will pass to *userret*. Function *userret* terminates the current process by calling *kill*.

Create also fills in the process table entry. In particular, *create* makes the state of the newly created process *PR_SUSP*, leaving it suspended, but otherwise ready to run. Finally, *create* returns the process ID of the newly created process; the process must be resumed before it can execute.

---

†Using a symbolic constant allows the choice to be overridden in a configuration file rather than requiring the code to be changed.

Many of the process initialization details depend on the C runtime environment as well as the calling conventions — one cannot write the code to start a process without understanding the details. For example, on an x86 platform, *create* arranges for arguments to be placed on the runtime stack; on an ARM platform, *create* places some arguments in registers and others on the stack. The code that pushes arguments may be difficult to understand because *create* copies the arguments directly from its own runtime stack onto the stack that it has allocated for the new process. To do so, it finds the address of the arguments on its own stack and moves through the list using pointer arithmetic. The following example shows how *create* forms the stack on an x86 platform.

```
/* create.c - create, newpid */

#include <xinu.h>

local   int newpid();

/*------------------------------------------------------------------------
 *  create  -  Create a process to start running a function on x86
 *------------------------------------------------------------------------
 */
pid32   create(
          void          *funcaddr,      /* Address of the function      */
          uint32        ssize,          /* Stack size in words          */
          pri16         priority,       /* Process priority > 0         */
          char          *name,          /* Name (for debugging)         */
          uint32        nargs,          /* Number of args that follow   */
          ...
        )
{
        uint32          savsp, *pushsp;
        intmask         mask;           /* Interrupt mask               */
        pid32           pid;            /* Stores new process id        */
        struct  procent *prptr;         /* Pointer to proc. table entry */
        int32           i;
        uint32          *a;             /* Points to list of args       */
        uint32          *saddr;         /* Stack address                */

        mask = disable();
        if (ssize < MINSTK)
                ssize = MINSTK;
        ssize = (uint32) roundmb(ssize);
        if ( (priority < 1) || ((pid=newpid()) == SYSERR) ||
             ((saddr = (uint32 *)getstk(ssize)) == (uint32 *)SYSERR) ) {
                restore(mask);
                return SYSERR;
        }
```

```
prcount++;
prptr = &proctab[pid];

/* Initialize process table entry for new process */
prptr->prstate = PR_SUSP;       /* Initial state is suspended   */
prptr->prprio = priority;
prptr->prstkbase = (char *)saddr;
prptr->prstklen = ssize;
prptr->prname[PNMLEN-1] = NULLCH;
for (i=0 ; i<PNMLEN-1 && (prptr->prname[i]=name[i])!=NULLCH; i++)
        ;
prptr->prsem = -1;
prptr->prparent = (pid32)getpid();
prptr->prhasmsg = FALSE;

/* Set up stdin, stdout, and stderr descriptors for the shell   */
prptr->prdesc[0] = CONSOLE;
prptr->prdesc[1] = CONSOLE;
prptr->prdesc[2] = CONSOLE;

/* Initialize stack as if the process was called                */

*saddr = STACKMAGIC;
savsp = (uint32)saddr;

/* Push arguments */
a = (uint32 *)(&nargs + 1);     /* Start of args                */
a += nargs -1;                  /* Last argument                */
for ( ; nargs > 0 ; nargs--)    /* Machine dependent; copy args */
        *--saddr = *a--;        /*   onto created process' stack*/
*--saddr = (long)INITRET;       /* Push on return address       */

/* The following entries on the stack must match what ctxsw     */
/*   expects a saved process state to contain: ret address,     */
/*   ebp, interrupt mask, flags, registers, and an old SP       */

*--saddr = (long)funcaddr;      /* Make the stack look like it's*/
                                /*   half-way through a call to */
                                /*   ctxsw that "returns" to the*/
                                /*   new process                */
*--saddr = savsp;               /* This will be register ebp    */
                                /*   for process exit           */
savsp = (uint32) saddr;         /* Start of frame for ctxsw     */
*--saddr = 0x00000200;          /* New process runs with        */
                                /*   interrupts enabled         */
```

```
        /* Basically, the following emulates an x86 "pushal" instruction*/

        *--saddr = 0;                   /* %eax */
        *--saddr = 0;                   /* %ecx */
        *--saddr = 0;                   /* %edx */
        *--saddr = 0;                   /* %ebx */
        *--saddr = 0;                   /* %esp; value filled in below  */
        pushsp = saddr;                 /* Remember this location       */
        *--saddr = savsp;               /* %ebp (while finishing ctxsw) */
        *--saddr = 0;                   /* %esi */
        *--saddr = 0;                   /* %edi */
        *pushsp = (unsigned long) (prptr->prstkptr = (char *)saddr);
        restore(mask);
        return pid;
}

/*------------------------------------------------------------------------
 *  newpid  -  Obtain a new (free) process ID
 *------------------------------------------------------------------------
 */
local   pid32   newpid(void)
{
        uint32  i;                      /* Iterate through all processes*/
        static  pid32 nextpid = 1;      /* Position in table to try or  */
                                        /*   one beyond end of table    */

        /* Check all NPROC slots */

        for (i = 0; i < NPROC; i++) {
                nextpid %= NPROC;       /* Wrap around to beginning */
                if (proctab[nextpid].prstate == PR_FREE) {
                        return nextpid++;
                } else {
                        nextpid++;
                }
        }
        return (pid32) SYSERR;
}
```

As described above, *create* arranges for a process to invoke function *userret* if the process returns from the top-level function. In fact, *create* stores the address of *userret* in the stack location where a return address would normally appear (using the symbolic constant *INITRET*). Placing the address of a function in the return address field is allowed because a function return merely jumps to the address. File *userret.c* contains the code.

```
/* userret.c - userret */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  userret  -  Called when a process returns from the top-level function
 *------------------------------------------------------------------------
 */
void    userret(void)
{
        kill(getpid());                 /* Force process to exit */
}
```

Create introduces an initial transition in the state diagram: a newly created process starts in the *suspended* state. Figure 6.3 illustrates the augmented state diagram.

Figure 6.3 The state diagram showing an initial transition to the *suspended* state.

6.13 Other Process Manager Functions

Three additional system calls help manage processes: *getpid*, *getprio*, and *chprio*. As we have seen, *getpid* allows the current process to obtain its process ID, and *getprio* allows a caller to obtain the scheduling priority of an arbitrary process. Another useful system call, *chprio*, allows a process to change the priority of an arbitrary process. The implementation of each of the three functions appears to be completely straightforward. For example, consider the code for *getprio*. After checking its argument, *getprio* ex-

tracts the scheduling priority for the specified process from the process table entry, and returns the priority to the caller.

```
/* getprio.c - getprio */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  getprio  -  Return the scheduling priority of a process
 *------------------------------------------------------------------------
 */
syscall getprio(
          pid32         pid             /* Process ID                   */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        uint32  prio;                   /* Priority to return           */

        mask = disable();
        if (isbadpid(pid)) {
                restore(mask);
                return SYSERR;
        }
        prio = proctab[pid].prprio;
        restore(mask);
        return prio;
}
```

Because global variable *currpid* contains the ID of the currently executing process, the code for *getpid* is trivial:

```
/* getpid.c - getpid */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  getpid  -  Return the ID of the currently executing process
 *------------------------------------------------------------------------
 */
pid32   getpid(void)
{
        return (currpid);
}
```

Function *chprio* allows the scheduling priority of any process to be changed. The code is found in *chprio.c*.

```
/* chprio.c - chprio */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  chprio  -  Change the scheduling priority of a process
 *------------------------------------------------------------------------
 */
pri16   chprio(
          pid32         pid,            /* ID of process to change      */
          pri16         newprio         /* New priority                 */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  procent *prptr;         /* Ptr to process' table entry  */
        pri16   oldprio;                /* Priority to return           */

        mask = disable();
        if (isbadpid(pid)) {
                restore(mask);
                return (pri16) SYSERR;
        }
        prptr = &proctab[pid];
        oldprio = prptr->prprio;
        prptr->prprio = newprio;
        restore(mask);
        return oldprio;
}
```

The implementation of *chprio* seems to do exactly what is needed. It checks to be sure the specified process exists before changing the priority field in its process table entry. As the exercises point out, however, the code contains two omissions.

6.14 Summary

The chapter expands the support for concurrent execution by adding a layer of process management software on top of a scheduler and context switch. The new layer includes routines to *suspend* and *resume* execution as well as routines that *create* a new process or *kill* an existing process. The chapter also examines three additional functions that obtain the ID of the current process (*getpid*), the scheduling priority of the current

process (*getprio*), or change the scheduling priority of an arbitrary process (*chprio*). Despite its brevity, the code built thus far forms the basis of a process manager. With proper initialization and support routines, our basic process manager can multiplex a processor among multiple concurrent computations.

Function *create* forms a new process, and leaves the process in the *suspended* state. *Create* allocates a stack for the new process, and places values on the stack and in the process table such that *ctxsw* can switch to the process and begin execution. The initial values are arranged in a pseudo call, as if the process was called from *userret*. If the process returns from its top-level function, control passes to *userret*, which calls *kill* to terminate the process.

EXERCISES

6.1 As the text suggests, a process can tell which of several events triggered resumption if its priority is set to a unique value for each separate call to *resume*. Use the method to create a process that suspends itself and determines which of two other processes resumes it first.

6.2 Suppose a system contains three processes, *A*, *B*, and *C*, with equal priority. If process *A* is executing and suspends *C*, which process will run? Explain.

6.3 Suppose a system contains three processes, *A*, *B*, and *C*, priorities 20, 20, and 10, and process *C* has been suspended. If process *A* is executing and resumes process *C*, which process will run? Why?

6.4 Why does *create* build a pseudo-call that returns to *userret* at process exit instead of one that calls *kill* directly?

6.5 Global variable *prcount* tells the number of active user processes. Carefully consider the code in *kill* and tell whether the count in *prcount* includes the null process?

6.6 When a process kills itself, *kill* deallocates the stack and *then* calls *resched*, which means the process continues to use the deallocated stack. Redesign the system so a current process does not deallocate its own stack, but instead moves to a new state, *PR_DYING*. Arrange for whatever process searches the process table to look for dying processes, free the stack, and move the entry to *PR_FREE*.

6.7 As the text mentions, *kill* calls *xdone* before the last process has been terminated. Change the system so the null process continuously monitors the count of user processes and calls *xdone* when all processes complete.

6.8 In the previous exercise, what restrictions does the new implementation impose on *xdone* that were not in the current implementation?

6.9 Some hardware architectures use a special instruction to allow an application program to invoke a system call. Investigate such an architecture, and describe exactly how a system call passes to the correct operating system function.

6.10 *Create* leaves the new process suspended instead of running. Why?

6.11 Function *resume* saves the resumed process's priority in a local variable before calling *ready*. Show that if it references *prptr->prprio* after the call to *ready*, *resume* can return a priority value that the resumed process never had (not even after resumption).

6.12 In function *newpid*, variable *nextpid* is a static integer that tells the next process table slot to check for a free one. Starting the search from where it left off eliminates looking past the used slots again and again. Speculate on whether the technique is worthwhile in an embedded system.

6.13 Function *chprio* contains two design flaws. The first arises because the code does not ensure that the new priority value is a positive integer. Describe what can happen if the priority of a process is set to –1.

6.14 The second design flaw in *chprio* violates a fundamental design principle. Identify the flaw, describe its consequences, and repair it.

6.15 In addition to the functionality discussed in the chapter, system calls are also responsible for enforcing system security policies. Choose an operating system and find out how system calls enforce security.

Chapter Contents

7.1 Introduction, 123
7.2 The Need For Synchronization, 123
7.3 A Conceptual View Of Counting Semaphores, 125
7.4 Avoidance Of Busy Waiting, 125
7.5 Semaphore Policy And Process Selection, 126
7.6 The Waiting State, 127
7.7 Semaphore Data Structures, 128
7.8 The Wait System Call, 129
7.9 The Signal System Call, 130
7.10 Static And Dynamic Semaphore Allocation, 131
7.11 Example Implementation Of Dynamic Semaphores, 132
7.12 Semaphore Deletion, 133
7.13 Semaphore Reset, 135
7.14 Coordination Across Parallel Processors (Multicore), 136
7.15 Perspective, 137
7.16 Summary, 137

7

Coordination Of Concurrent Processes

The future belongs to him who knows how to wait.

— Russian Proverb

7.1 Introduction

Previous chapters introduce pieces of a process manager, including scheduling, context switching, and functions that create and terminate processes. This chapter continues the exploration of process management by discussing functions that a set of processes can use to coordinate and synchronize their actions. The chapter explains the motivation for such primitives and their implementation. The chapter also considers coordination of multiple processors, such as those on a multicore chip.

The next chapter extends our discussion of a process manager by describing a low-level message passing mechanism. Later chapters show how synchronization functions are used to perform I/O.

7.2 The Need For Synchronization

Because they execute concurrently, processes need to cooperate when sharing global resources. In particular, an operating system designer must ensure that only one process attempts to change a given variable at any time. For example, consider the process table. When a new process is created, a slot in the table must be allocated and

values inserted. If two processes each attempt to create a new process, the system must guarantee that only one of them can execute *create* at a given time, or errors can result.

The previous chapter illustrates one approach system functions can take to guarantee that no other process interferes with them: a function disables interrupts and avoids using any functions that call *resched*. Indeed, system calls such as *suspend*, *resume*, *create*, and *kill* each use the approach.

Why not use the same solution whenever a process needs to guarantee non-interference? The answer is that disabling interrupts has an undesirable global effect on all parts of the system: it stops all activity except for one process, and limits what the process can do. In particular, no I/O can occur while interrupts are disabled. We will learn later that disabling interrupts too long can cause problems (e.g., if packets continue to arrive over a network while interrupts are disabled, the network interface will start to discard them). Therefore, we need a general-purpose coordination mechanism that permits arbitrary subsets of the processes to coordinate the use of individual data items without disabling device interrupts for long periods of time, without interfering with processes outside the subset, and without limiting what the running process can do. For example, it should be possible for one process to prohibit changes to a large data structure long enough to format and print the data, without stopping processes that do not need to access the data structure. The mechanism should be transparent: a programmer should be able to understand the consequences of process coordination. Thus, further synchronization mechanisms are needed that:

- Allow a subset of processes to contend for access to a resource
- Provide a policy that guarantees fair access

The first item ensures that coordination is local: instead of disabling all interrupts, only those processes contending for a given resource will block waiting for access. Other parts of the system can continue to operate unaffected. The second item ensures that if K processes all attempt to access a given resource, each of the K will eventually receive access (i.e., no process is *starved*).

Chapter 2 introduces the fundamental mechanism that solves the problem: *counting semaphores*. The chapter also provides examples that show how processes use semaphores to coordinate. As Chapter 2 indicates, semaphores provide an elegant solution for two problems:

- Mutual exclusion
- Producer–consumer interaction

Mutual exclusion. The term *mutual exclusion* is used to describe a situation where a set of processes needs to guarantee that only one of them operates at a given time. Mutual exclusion includes access to shared data, but can also include access to an arbitrary shared resource, such as an I/O device.

Producer–consumer interaction. We use the term *producer–consumer interaction* to refer to a situation where processes exchange data items. In the simplest form, one process acts as a *producer* by generating a sequence of data items, and another process acts as a *consumer* by accepting the data items. In more complex forms, one or more processes can act as producers and one or more processes can act as consumers. The key to coordinating the interaction is that each item produced must be received by exactly one consumer (i.e., no items are lost and no items are duplicated).

Both forms of process coordination arise throughout an operating system. For example, consider a set of applications that are producing messages to be displayed on the console. The console device software must coordinate processes to ensure that characters do not arrive faster than the hardware can display them. Outgoing characters can be placed in a buffer in memory. Once the buffer fills, the producer must be blocked until space becomes available. Similarly, if the buffer becomes empty, the device stops sending characters. The key idea is that a producer must be blocked when the consumer is not ready to receive data, and a consumer must be blocked when a producer is not ready to send data.

7.3 A Conceptual View Of Counting Semaphores

A counting semaphore mechanism that solves both problems described above has a surprisingly elegant implementation. Conceptually, a semaphore, *s*, consists of an integer count and a set of blocked processes. Once a semaphore has been created, processes use two functions, *wait* and *signal*, to operate on the semaphore. A process calls *wait*(*s*) to decrement the count of semaphore *s*, and *signal*(*s*) to increment the count. If the semaphore count becomes negative when a process executes *wait*(*s*), the process is temporarily blocked and placed in the semaphore's set of blocked processes. From the point of view of the process, the call to *wait* does not return for a while. A blocked process becomes ready to run again when another process calls *signal* to increment the semaphore count. That is, if any processes are blocked waiting for a semaphore when *signal* is called, one of the blocked processes will be made ready and allowed to execute. Of course, a programmer must use semaphores with caution: if no process ever signals the semaphore, the blocked processes will wait forever.

7.4 Avoidance Of Busy Waiting

What should a process do while waiting on a semaphore? It might seem that after it decrements the semaphore count, a process could repeatedly test the count until the value becomes positive. On a single processor system, however, such *busy waiting* is unacceptable because other processes will be deprived of the processor. If no other process receives processor service, no process can call *signal* to terminate the wait. Therefore, operating systems avoid busy waiting. Instead, semaphore implementations follow an important principle:

While a process waits on a semaphore, the process does not execute instructions.

7.5 Semaphore Policy And Process Selection

To implement semaphores without busy waiting, an operating system associates a process list with each semaphore. Only the current process can choose to wait on a semaphore. When a process waits on semaphore *s*, the system decrements the count associated with *s*. If the count becomes negative, the process must be blocked. To block a process, the system places the process on the list associated with the semaphore, changes the state so the process is no longer current, and calls *resched* to allow other processes to run.

Later, when *signal* is called on semaphore *s*, the semaphore count is incremented. In addition, the *signal* examines the process list associated with *s*. If the list is not empty (i.e., at least one process is waiting on the semaphore), *signal* extracts a process from the list and moves the process back to the ready list.

A question arises: if multiple processes are waiting, which one should *signal* select? Several policies have been used:

- Highest scheduling priority
- First-come-first-served (longest waiting time)
- Random

Although it may seem reasonable, selecting the highest priority waiting process violates the principle of fairness. To see why, consider a set of low-priority and high-priority processes that are using a mutual exclusion semaphore. Suppose each process repeatedly waits on the semaphore, uses the resource, and signals the semaphore. If the semaphore system always selects a high-priority process and the scheduling policy always gives the processor to high-priority processes, the low-priority processes can be blocked forever while high-priority processes continue to gain access.

To avoid unfairness, many implementations choose a first-come-first-served policy: if processes are waiting, the system always chooses the process that has been waiting the longest. The implementation of a first-come-first-served policy is both elegant and efficient: the system creates a FIFO queue for each semaphore, and uses the queue to store processes that are waiting. When it needs to block a process, *wait* inserts the process at the tail; when it needs to unblock a process, *signal* extracts a process from the head.

A first-come-first-served policy can lead to a *priority inversion* in the sense that a high-priority process can be blocked on a semaphore while a low-priority process executes. In addition, it can lead to a synchronization problem discussed in an exercise. One alternative consists of choosing among waiting processes at *random*. The chief disadvantage of random selection lies in computational overhead (e.g., random number generation).

After considering the advantages and disadvantages of various schemes, we chose a first-come-first-served policy for Xinu:

> *Xinu semaphore process selection policy: if one or more processes are waiting for semaphore* s *when a* signal *operation occurs for* s*, the process that has been waiting the longest becomes ready.*

7.6 The Waiting State

In what state should a process be placed while it is waiting for a semaphore? Because it is neither using the processor nor eligible to run, the process is neither *current* nor *ready*. The *suspended* state, introduced in the previous chapter cannot be used because functions *suspend* and *resume*, which move processes in and out of the suspended state, have no connection with semaphores. More important, processes waiting for semaphores appear on a list, but suspended processes do not — *kill* must distinguish the two cases when terminating a process. Because existing states do not adequately encompass processes waiting on a semaphore, a new state must be invented. We call the new state *waiting*, and use symbolic constant *PR_WAIT* in the code. Figure 7.1 shows the expanded state transition diagram.

Figure 7.1 State transitions including the *waiting* state.

7.7 Semaphore Data Structures

The example system stores semaphore information in a global semaphore table, *semtab*. Each entry in *semtab* corresponds to one semaphore. An entry contains an integer count for the semaphore, and the ID of a queue that can be used to hold waiting processes. The definition of an entry is given by structure *sentry*. File *semaphore.h* contains the details.

```
/* semaphore.h - isbadsem */

#ifndef NSEM
#define NSEM            120     /* Number of semaphores, if not defined */
#endif

/* Semaphore state definitions */

#define S_FREE  0               /* Semaphore table entry is available   */
#define S_USED  1               /* Semaphore table entry is in use      */

/* Semaphore table entry */
struct  sentry  {
        byte    sstate;         /* Whether entry is S_FREE or S_USED    */
        int32   scount;         /* Count for the semaphore              */
        qid16   squeue;         /* Queue of processes that are waiting  */
                                /*     on the semaphore                 */
};

extern  struct  sentry semtab[];

#define isbadsem(s)     ((int32)(s) < 0 || (s) >= NSEM)
```

In structure *sentry*, field *scount* contains the current integer count of the semaphore. The list of processes waiting for a semaphore resides in the queue structure, and field *squeue* gives the index of the head of the list for a given semaphore. The state field, *sstate*, tells whether the entry is currently used (i.e., allocated) or free (currently unallocated).

Throughout the system, semaphores are identified by an integer ID. As with other identification values, semaphore IDs are assigned to make lookup efficient: the semaphore table is an array, and each ID is an index in the array. To summarize:

> *A semaphore is identified by its index in the global semaphore table,* semtab.

7.8 The Wait System Call

Recall that the two primary operations on a semaphore are *wait* and *signal*. *Wait* decrements the count of a semaphore. If the count remains nonnegative, *wait* returns to the caller immediately. In essence, a process executing *wait* on a semaphore with a nonpositive count voluntarily gives up control of the processor. That is, *wait* enqueues the calling process on the list for the semaphore, changes the process state to *PR_WAIT*, and calls *resched* to switch to a ready process. Also recall that our policy maintains the list of processes as a FIFO queue, which means a new process is inserted at the tail of a list. File *wait.c* contains the code.

```
/* wait.c - wait */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  wait  -  Cause current process to wait on a semaphore
 *------------------------------------------------------------------------
 */
syscall wait(
          sid32         sem             /* Semaphore on which to wait  */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  procent *prptr;         /* Ptr to process' table entry  */
        struct  sentry *semptr;         /* Ptr to sempahore table entry */

        mask = disable();
        if (isbadsem(sem)) {
                restore(mask);
                return SYSERR;
        }

        semptr = &semtab[sem];
        if (semptr->sstate == S_FREE) {
                restore(mask);
                return SYSERR;
        }

        if (--(semptr->scount) < 0) {           /* If caller must block */
                prptr = &proctab[currpid];
                prptr->prstate = PR_WAIT;       /* Set state to waiting */
                prptr->prsem = sem;             /* Record semaphore ID  */
                enqueue(currpid,semptr->squeue);/* Enqueue on semaphore */
                resched();                      /*   and reschedule     */
        }
```

```
        restore(mask);
        return OK;
}
```

Once enqueued on a semaphore list, a process remains in the waiting state (i.e., not eligible to execute) until the process reaches the head of the queue and some other process signals the semaphore. When the call to *signal* moves a waiting process back to the ready list, the process becomes eligible to use the processor, and eventually resumes execution. From the point of view of the waiting process, its last act consisted of a call to *ctxsw*. When the process restarts, the call to *ctxsw* returns to *resched*, the call to *resched* returns to *wait*, and the call to *wait* returns to the location from which it was called.

7.9 The Signal System Call

Function *signal* takes a semaphore ID as an argument, increments the count of the specified semaphore, and makes the first process ready, if any are waiting. Although it may seem difficult to understand why *signal* makes a process ready even though the semaphore count remains negative or why *wait* does not always enqueue the calling process, the reason is both easy to understand and easy to implement. *Wait* and *signal* maintain the following invariant regarding the count of a semaphore:

> *Semaphore invariant: a nonnegative semaphore count means that the queue is empty; a semaphore count of negative* N *means that the queue contains* N *waiting processes.*

In essence, a count of positive *N* means that *wait* can be called *N* more times before any process blocks. Because *wait* and *signal* each change the semaphore count, they must each adjust the queue length to reestablish the invariant. When it decrements the count, *wait* examines the result, and adds the current process to the queue if the new count is negative. Because it increments the count, *signal* examines the queue and removes a process from the queue if the queue is nonempty.

```
/* signal.c - signal */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  signal  -  Signal a semaphore, releasing a process if one is waiting
 *------------------------------------------------------------------------
 */
```

```
syscall signal(
          sid32         sem             /* ID of semaphore to signal    */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  sentry *semptr;         /* Ptr to sempahore table entry */

        mask = disable();
        if (isbadsem(sem)) {
                restore(mask);
                return SYSERR;
        }
        semptr= &semtab[sem];
        if (semptr->sstate == S_FREE) {
                restore(mask);
                return SYSERR;
        }
        if ((semptr->scount++) < 0) {   /* Release a waiting process */
                ready(dequeue(semptr->squeue));
        }
        restore(mask);
        return OK;
}
```

7.10 Static And Dynamic Semaphore Allocation

An operating system designer must choose between two approaches for semaphore allocation:

- Static allocation: a programmer defines a fixed set of semaphores at compile time; the set does not change as the system runs.
- Dynamic allocation: the system includes functions that allow semaphores to be created on demand and deallocated when they are no longer needed.

The advantage of static allocation lies in saving space and reducing processing overhead — the system only contains memory for the needed semaphores, and the system does not require functions to allocate or deallocate semaphores. Thus, the smallest embedded systems use static allocation.

The chief advantage of dynamic allocation arises from the ability to accommodate new uses at runtime. For example, a dynamic allocation scheme allows a user to launch an application that allocates a semaphore, terminate the application, and then launch another application. Thus, larger embedded systems and most large operating systems provide dynamic allocation of resources, including semaphores. The next sections show that dynamic allocation does not introduce much additional code.

7.11 Example Implementation Of Dynamic Semaphores

Xinu provides a limited form of dynamic allocation: processes can create semaphores dynamically, and a given process can create multiple semaphores, provided the total number of semaphores allocated simultaneously does not exceed a predefined maximum. Furthermore, to minimize the allocation overhead, the system preallocates a list in the queue structure for each semaphore when the operating system boots. Thus, only a small amount of work needs be done when a process creates a semaphore.

Two system calls, *semcreate* and *semdelete*, handle dynamic semaphore allocation and deallocation. *Semcreate* takes an initial semaphore count as an argument, allocates a semaphore, assigns the semaphore the specified count, and returns the semaphore ID. To preserve the semaphore invariant, the initial count must be nonnegative. Therefore, *semcreate* begins by testing its argument. If the argument is valid, *semcreate* searches the semaphore table, *semtab*, for an unused entry and initializes the count. To search the table, *semcreate* calls function *newsem*, which iterates through all *NSEM* entries of the table. If no free entry is found, *newsem* returns *SYSERR*. Otherwise, *newsem* changes the state of the entry to *S_USED*, and returns the table index as the ID.

Once a table entry has been allocated, *semcreate* only needs to initialize the count and return the index of the semaphore to its caller; the head and tail of a queue used to store waiting processes have been allocated when the operating system boots. File *semcreate.c* contains the code for function *newsem* as well as function *semcreate*. Note the use of a static index variable *nextsem* to optimize searching (i.e., allow a search to start where the last search left off).

```
/* semcreate.c - semcreate, newsem */

#include <xinu.h>

local   sid32   newsem(void);

/*------------------------------------------------------------------------
 *  semcreate  -  Create a new semaphore and return the ID to the caller
 *------------------------------------------------------------------------
 */
sid32   semcreate(
          int32         count           /* Initial semaphore count      */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        sid32   sem;                    /* Semaphore ID to return       */

        mask = disable();

        if (count < 0 || ((sem=newsem())==SYSERR)) {
```

```
                restore(mask);
                return SYSERR;
        }
        semtab[sem].scount = count;     /* Initialize table entry       */

        restore(mask);
        return sem;
}

/*------------------------------------------------------------------------
 *  newsem  -  Allocate an unused semaphore and return its index
 *------------------------------------------------------------------------
 */
local   sid32   newsem(void)
{
        static  sid32   nextsem = 0;    /* Next semaphore index to try  */
        sid32   sem;                    /* Semaphore ID to return       */
        int32   i;                      /* Iterate through # entries    */

        for (i=0 ; i<NSEM ; i++) {
                sem = nextsem++;
                if (nextsem >= NSEM)
                        nextsem = 0;
                if (semtab[sem].sstate == S_FREE) {
                        semtab[sem].sstate = S_USED;
                        return sem;
                }
        }
        return SYSERR;
}
```

7.12 Semaphore Deletion

Function *semdelete* reverses the actions of *semcreate*. *Semdelete* takes the ID of a semaphore as an argument and releases the semaphore table entry for subsequent use. Deallocating a semaphore requires three steps. First, *semdelete* verifies that the argument specifies a valid semaphore ID and that the corresponding entry in the semaphore table is currently in use. Second, *semdelete* sets the state of the entry to *S_FREE* to indicate that the table entry can be reused. Finally, *semdelete* iterates through the set of processes that are waiting on the semaphore and makes each process ready. File *semdelete.c* contains the code.

```
/* semdelete.c - semdelete */

#include <xinu.h>

/*------------------------------------------------------------------------
 * semdelete  -  Delete a semaphore by releasing its table entry
 *------------------------------------------------------------------------
 */
syscall semdelete(
          sid32         sem             /* ID of semaphore to delete    */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  sentry *semptr;         /* Ptr to semaphore table entry */

        mask = disable();
        if (isbadsem(sem)) {
                restore(mask);
                return SYSERR;
        }

        semptr = &semtab[sem];
        if (semptr->sstate == S_FREE) {
                restore(mask);
                return SYSERR;
        }
        semptr->sstate = S_FREE;

        resched_cntl(DEFER_START);
        while (semptr->scount++ < 0) {  /* Free all waiting processes   */
                ready(getfirst(semptr->squeue));
        }
        resched_cntl(DEFER_STOP);
        restore(mask);
        return OK;
}
```

If processes remain enqueued on a semaphore when the semaphore is deallocated, an operating system must handle each of the processes. In the example implementation, *semdelete* places each waiting process back on the ready list, allowing the process to resume execution as if the semaphore had been signaled. The example only represents one strategy, and other strategies are possible. For example, some operating systems consider it an error to attempt to deallocate a semaphore on which processes are waiting. The exercises suggest exploring alternatives.

Note that the code to make processes ready uses deferred rescheduling. That is, *semdelete* calls *resched_cntl* to start deferral before making processes ready, and only calls *resched_cntl* to end the deferral period after all waiting processes have been moved to the ready list. The second call will invoke *resched* to reestablish the scheduling invariant.

7.13 Semaphore Reset

It is sometimes convenient to reset the count of a semaphore without incurring the overhead of deleting an existing semaphore and creating a new one. The system call *semreset*, shown in file *semreset.c* below, resets the count of a semaphore.

```
/* semreset.c - semreset */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  semreset  -  Reset a semaphore's count and release waiting processes
 *------------------------------------------------------------------------
 */
syscall semreset(
          sid32         sem,            /* ID of semaphore to reset     */
          int32         count           /* New count (must be >= 0)     */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  sentry *semptr;         /* Ptr to semaphore table entry */
        qid16   semqueue;               /* Semaphore's process queue ID */
        pid32   pid;                    /* ID of a waiting process      */

        mask = disable();

        if (count < 0 || isbadsem(sem) || semtab[sem].sstate==S_FREE) {
                restore(mask);
                return SYSERR;
        }

        semptr = &semtab[sem];
        semqueue = semptr->squeue;      /* Free any waiting processes */
        resched_cntl(DEFER_START);
        while ((pid=getfirst(semqueue)) != EMPTY)
                ready(pid);
        semptr->scount = count;         /* Reset count as specified */
```

```
        resched_cntl(DEFER_STOP);
        restore(mask);
        return OK;
}
```

Semreset must preserve the semaphore invariant. Rather than build a general-purpose solution that allows a caller to specify an arbitrary semaphore count, our implementation takes a simplified approach by requiring the new count to be nonnegative. As a result, once the semaphore count has been changed, the queue of waiting processes will be empty. As with *semdelete*, *semreset* must be sure that no processes are already waiting on the semaphore. Thus, after checking its arguments and verifying that the semaphore exists, *semreset* iterates through the list of waiting processes, removing each from the semaphore queue and making the process ready to execute. As expected, *semreset* uses *resched_cntl* to defer rescheduling during the time processes are placed on the ready list.

7.14 Coordination Across Parallel Processors (Multicore)

The semaphore system described above works well on a computer that has a single processor core. However, many modern processor chips include multiple cores. One core is usually dedicated to run operating system functions, and other cores are used to execute user applications. On such systems, using semaphores supplied by the operating system to coordinate processes can be inefficient. To see why, consider what happens when an application running on core 2 needs exclusive access to a specific memory location. The application process calls *wait*, which must pass the request to the operating system on core 1. Communication among cores often involves raising an interrupt. Furthermore, while it runs an operating system function, core 1 disables interrupts, which defeats one of the reasons to use semaphores.

Some multiprocessor systems supply hardware primitives, known as *spin locks*, that allow multiple processors to contend for mutually exclusive access. The hardware defines a set of K spin locks (K might be less than 1024). Conceptually, each of the spin locks is a single bit, and the spin locks are initialized to zero. The instruction set includes a special instruction, called a *test-and-set*, that a core can use to coordinate. A test-and-set performs two operations atomically: it sets a spin lock to 1 and returns the value of the spin lock before the operation. The hardware guarantees atomicity, which means that if two or more processors attempt to set a given spin lock simultaneously, one of them will receive 0 as the previous value and the others will receive 1. Once it finishes using the locked item, the core that obtained the lock resets the value to 0, allowing another core to obtain the lock.

To see how spin locks work, suppose two cores need exclusive access to a shared data item and are using spin lock 5. When a core wants to obtain mutually exclusive access, the core executes a loop:†

†Because it uses hardware instructions, test-and-set code is usually written in assembly language; it is shown in pseudo code for clarity.

```
while (test_and_set(5)) {
        ;
}
```

The loop repeatedly uses the test-and-set instruction to set spin lock 5. If the lock was set before the instruction executed, the instruction will return 1, and the loop will continue. If the lock was not set before the instruction executed, the hardware will return 0, and the loop terminates. If multiple processors are all trying to set spin lock 5 at the same time, the hardware guarantees that only one will be granted access. Thus, test_and_set is analogous to *wait*.

Once a core finishes using the shared data, the core executes an instruction that clears the spin lock:

```
clear(5);
```

On multicore machines, vendors include various instructions that can be used as a spin lock. For example, in addition to *test-and-set*, Intel multicore processors provide an atomic *compare-and-swap* instruction. If multiple cores attempt to execute a compare-and-swap instruction at the same time, one of them will succeed and the others will all find that the comparison fails. A programmer can use such instructions to build the equivalent of a spin lock.

It may seem that a spin lock is wasteful because a processor merely blocks (i.e., busy waits) in a loop until access is granted. However, if the probability of two processors contending for access is low, a spin lock mechanism is much more efficient than a system call (e.g., waiting on a semaphore). Therefore, a programmer must be careful in choosing when to use spin locks and when to use system calls.

7.15 Perspective

The counting semaphore abstraction is significant for two reasons. First, it provides a powerful mechanism that can be used to control both mutual exclusion and producer–consumer synchronization, the two primary process coordination paradigms. Second, the implementation is surprisingly compact and extremely efficient. To appreciate the small size, reconsider functions *wait* and *signal*. If the code to test arguments and returns results is removed, only a few lines of code remain. As we examine the implementation of other abstractions, the point will become more significant: despite their importance, only a trivial amount of code is needed to implement counting semaphores.

7.16 Summary

Instead of disabling interrupts, which stops all activities other than the current process, operating systems offer synchronization primitives that allow subsets of processes to coordinate without affecting other processes. A fundamental coordination mechanism, known as a counting semaphore, allows processes to coordinate without us-

ing busy-waiting. Each semaphore consists of an integer count plus a queue of processes. The semaphore adheres to an invariant that specifies a count of negative *N* means the queue contains *N* processes.

The two fundamental primitives, *signal* and *wait*, permit a caller to increment or decrement the semaphore count. If a call to *wait* makes the semaphore count negative, the calling process is placed in the *waiting* state and the processor passes to another process. In essence, a process that waits for a semaphore voluntarily enqueues itself on the list of processes waiting for the semaphore, and calls *resched* to allow other processes to execute.

Either static or dynamic allocation can be used with semaphores. The example code includes functions *semcreate* and *semdelete* to permit dynamic allocation. If a semaphore is deallocated while processes are waiting, the processes must be handled. The example code makes the processes ready as if the semaphore had been signaled.

Multiprocessors can use a mutual exclusion mechanism known as spin locks. Although spin locks seem inefficient because they require a processor to repeatedly test for access, they can be more efficient than an arrangement where one processor interrupts another to place a system call.

EXERCISES

7.1 The text notes that some operating systems consider semaphore deletion to be in error if processes remain enqueued waiting for the semaphore. Rewrite *semdelete* to return *SYSERR* for a busy semaphore.

7.2 As an alternative to the semaphore deletion mechanism illustrated in the chapter, consider using deferred deletion. That is, rewrite *semdelete* to place a deleted semaphore in a *deferred* state until all processes have been signaled. Modify *signal* to release the semaphore table entry when the last waiting process has been removed from the queue.

7.3 In the previous exercise, can deferred deletion have unexpected side effects? Explain.

7.4 As a further alternative to the deferred deletion of an active semaphore, modify *wait* to return a value *DELETED* if the semaphore was deleted while the calling process was waiting. (Choose a value for *DELETED* that differs from *SYSERR* and *OK*.) How can a process determine whether the semaphore on which it was waiting has been deleted? Be careful: remember that a high-priority process can execute at any time. Thus, after a low-priority process becomes ready, a higher-priority process can obtain the processor and create a new semaphore that reuses the semaphore table entry before the low-priority process completes execution of *wait*. Hint: consider adding a sequence field to the semaphore table entry.

7.5 Instead of allocating a central semaphore table, arrange to have each process allocate space for semaphore entries as needed, and use the address of an entry as the semaphore ID. Compare the approach to the centralized table in the example code. What are the advantages and disadvantages of each?

7.6 *Wait*, *signal*, *semcreate*, and *semdelete* coordinate among themselves for use of the semaphore table. Is it possible to use a semaphore to protect use of the semaphore table? Explain.

7.7 Consider a possible optimization: instead of using *ready*, arrange for *semdelete* to examine the priority of each waiting process before the process is placed on the ready list. If none of the processes has higher priority than the current process, do not reschedule, but if any of them has a higher priority, call *resched*. What is the cost of the optimization and what is the potential savings?

7.8 Construct a new system call, *signaln*(*sem*, *n*) that signals semaphore *sem n* times. Can you find an implementation that is more efficient than *n* calls to *signal*? Explain.

7.9 The example code uses a FIFO policy for semaphores. That is, when a semaphore is signaled, the process that has been waiting the longest becomes ready. Imagine a modification in which the processes waiting for a semaphore are kept on a priority queue ordered by process priority (i.e., when a semaphore is signaled, the highest priority waiting process becomes ready). What is the chief disadvantage of a priority approach?

7.10 Languages meant specifically for writing concurrent programs often have coordination and synchronization embedded in the language constructs directly. For example, it might be possible to declare functions in groups such that the compiler automatically inserts code to prohibit more than one process from executing a given group at a given time. Find an example of a language designed for concurrent programming, and compare process coordination with the semaphores in the Xinu code.

7.11 When a programmer is required to manipulate semaphores explicitly, what types of mistakes can a programmer make?

7.12 When it moves a waiting process to the ready state, *wait* sets field *prsem* in the process table entry to the ID of the semaphore on which the process is waiting. Will the value ever be used?

7.13 If a programmer makes a mistake, it is more likely that the error will produce 0 or 1 than an arbitrary integer. To help prevent errors, change *newsem* to begin allocating semaphores from the high end of the table, leaving slots 0 and 1 unused until all other entries have been exhausted. Suggest better ways of identifying semaphores that increase the ability to detect errors.

7.14 Function *semdelete* behaves in an unexpected way when deleting a semaphore with a non-negative count. Identify the behavior and rewrite the code to correct it.

7.15 Draw a call graph of all operating system functions from Chapters 4 through 7, showing which functions a given function invokes. Can a multi-level structure be deduced from the graph? Explain.

Chapter Contents

8.1 Introduction, 143
8.2 Two Types Of Message Passing Services, 143
8.3 Limits On Resources Used By Messages, 144
8.4 Message Passing Functions And State Transitions, 145
8.5 Implementation Of Send, 146
8.6 Implementation Of Receive, 148
8.7 Implementation Of Non-Blocking Message Reception, 149
8.8 Perspective, 149
8.9 Summary, 150

8

Message Passing

The message of history is clear: the past lies before us.

— Anonymous

8.1 Introduction

Previous chapters explain the basic components of a process manager, including: scheduling, context switching, and counting semaphores that provide coordination among concurrent processes. The chapters show how processes are created and how they terminate, and explain how an operating system keeps information about each process in a central table.

This chapter concludes our examination of basic process management facilities. The chapter introduces the concept of message passing, describes possible approaches, and shows an example of a low-level message passing system. Chapter 11 explains how a high-level message passing facility can be built using the basic process management mechanisms.

8.2 Two Types Of Message Passing Services

We use the term *message passing* to refer to a form of inter-process communication in which one process transfers (usually a small amount of) data to another. In some systems, processes deposit and retrieve messages from named *pickup points* that are sometimes called *mailboxes*. In other systems, a message must be addressed directly to a process. Message passing is both convenient and powerful, and some operating systems use it as the basis for all communication and coordination among processes.

For example, an operation such as transmitting data across a computer network can be implemented using message passing primitives.

Some message passing facilities provide process coordination because the mechanism delays a receiver until a message arrives. Thus, message passing can replace process suspension and resumption. Can message passing also replace synchronization primitives such as semaphores? The answer depends on the implementation of message passing. There are two types:

- *Synchronous*. If a receiver attempts to receive a message before the message has arrived, the receiver blocks; if a sender tries to send a message before a receiver is ready, the sender blocks. Sending and receiving processes must coordinate or one can become blocked waiting for the other.
- *Asynchronous*. A message can arrive at any time, and a receiver is notified. A receiver does not need to know in advance how many messages will arrive or how many senders will send messages.

Although it may lack generality and convenience, a synchronous message passing facility can serve in place of a semaphore mechanism. For example, consider a producer–consumer paradigm. Each time it generates new data, a producer process can send a message to the consumer process. Similarly, instead of waiting on a semaphore, the consumer can wait for a message. Using message passing to implement mutual exclusion is more complex, but usually possible.

The chief advantage of a synchronous message passing system arises because it fits well with a traditional computational paradigm. To receive a message in a synchronous system, a process calls a system function, and the call does not return until a message arrives. In contrast, an asynchronous message passing system either requires a process to *poll* (i.e., check for a message periodically) or requires a mechanism that allows the operating system to stop a process temporarily, allows the process to handle a message, and then resumes normal execution. Although it introduces additional overhead or complexity, asynchronous message passing can be convenient if a process does not know how many messages it will receive, when the messages will be sent, or which processes will send messages.

8.3 Limits On Resources Used By Messages

Xinu supports two forms of message passing that illustrate a completely synchronous paradigm and a partially asynchronous paradigm. The two facilities also illustrate the difference between direct and indirect message delivery: one provides a direct exchange of messages among processes, and the other arranges for messages to be exchanged through rendezvous points. This chapter begins the discussion by examining a facility that provides direct communication from one process to another. Chapter 11 discusses a second message passing facility. Separating message passing into two in-

dependent pieces has the advantage of making low-level message passing among processes efficient, while allowing a programmer to choose a more complex rendezvous approach when needed.

The Xinu process-to-process message passing system has been designed carefully to ensure that a process does not block while sending a message, and waiting messages do not consume all of memory. To make such guarantees, the message passing facility follows three guidelines:

- *Limited message size*. The system limits each message to a small, fixed size. In our example code, each message consists of a single word (i.e., an integer or a pointer).
- *No message queues*. The system permits a given process to store only one unreceived message per process at any time. There are no message queues.
- *First message semantics*. If several messages are sent to a given process before the process receives any of them, only the first message is stored and delivered; subsequent senders do not block.

The concept of *first message semantics* makes the mechanism useful for determining which of several events completes first. A process that needs to wait for events can arrange for each event to send a unique message. The process then waits for a message, and the operating system guarantees that the process will receive the first message that is sent.

8.4 Message Passing Functions And State Transitions

Three system calls manipulate messages: *send*, *receive*, and *recvclr*. *Send* takes a message and a process ID as arguments, and delivers the message to the specified process. *Receive*, which does not require arguments, causes the current process to wait until a message arrives, and then returns the message to its caller. *Recvclr* provides a non-blocking version of *receive*. If the current process has received a message when *recvclr* is called, the call returns the message exactly like *receive*. If no message is waiting, however, *recvclr* returns the value *OK* to its caller immediately, without delaying to wait for a message to arrive. As the name implies, *recvclr* can be used to remove an old message before engaging in a round of message passing.

The question arises: in what state should a process be while waiting for a message? Because waiting for a message differs from being ready to execute, waiting for a semaphore, waiting for the processor, suspended animation, or current execution, none of the existing states suffices. Thus, another state must be added to our design. The new state, *receiving*, is denoted in the example software with the symbolic constant *PR_RECV*. Adding the state produces the transition diagram illustrated in Figure 8.1.

Figure 8.1 Illustration of state transitions including the *receiving* state.

8.5 Implementation Of Send

A message passing system requires agreement between senders and receivers because a sender must store a message in a location from which the receiver can extract the message. A message cannot be stored in the sender's memory because a sending process can exit before the message is received. Most operating systems do not permit a sender to place a message in a receiver's address space because allowing a process to write into the memory allocated to another process poses a security threat. In our example system, restrictions on the size of messages eliminate the problem. Our implementation reserves space for one message in field *prmsg* of the recipient's process table entry.

To deposit a message, function *send* first checks that the specified recipient process exists. It then checks to ensure the recipient does not have a message outstanding. To do so, *send* examines field *prhasmsg* in the recipient's process table entry. If the recipient has no outstanding message, *send* deposits the new message in the *prmsg* field and sets *prhasmsg* to *TRUE* to indicate that a message is waiting. As a final step, if the recipient is waiting for the arrival of a message (i.e., the recipient process has state *PR_RECV* or state *PR_RECTIM*), *send* calls *ready* to make the process ready and re-

establish the scheduling invariant. In the case of *PR_RECTIM*, which is discussed later in the text, *send* must first call *unsleep* to remove the process from the queue of sleeping processes. File *send.c* contains the code.

```
/* send.c - send */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  send  -  Pass a message to a process and start recipient if waiting
 *------------------------------------------------------------------------
 */
syscall send(
          pid32         pid,            /* ID of recipient process      */
          umsg32        msg             /* Contents of message          */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  procent *prptr;         /* Ptr to process' table entry  */

        mask = disable();
        if (isbadpid(pid)) {
                restore(mask);
                return SYSERR;
        }

        prptr = &proctab[pid];
        if ((prptr->prstate == PR_FREE) || prptr->prhasmsg) {
                restore(mask);
                return SYSERR;
        }
        prptr->prmsg = msg;             /* Deliver message              */
        prptr->prhasmsg = TRUE;         /* Indicate message is waiting  */

        /* If recipient waiting or in timed-wait make it ready */

        if (prptr->prstate == PR_RECV) {
                ready(pid);
        } else if (prptr->prstate == PR_RECTIM) {
                unsleep(pid);
                ready(pid);
        }
        restore(mask);          /* Restore interrupts */
        return OK;
}
```

8.6 Implementation Of Receive

A process calls *receive* (or *recvclr*) to obtain an incoming message. *Receive* examines the process table entry for the current process, and uses the *prhasmsg* field to determine whether a message is waiting. If no message has arrived, *receive* changes the process state to *PR_RECV*, and calls *resched*, to allow other processes to run. When another process sends the receiving process a message, the call to *resched* returns. Once execution passes the *if* statement, *receive* extracts the message, sets *prhasmsg* to *FALSE*, and returns the message to its caller. File *receive.c* contains the code:

```
/* receive.c - receive */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  receive  -  Wait for a message and return the message to the caller
 *------------------------------------------------------------------------
 */
umsg32  receive(void)
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  procent *prptr;         /* Ptr to process' table entry  */
        umsg32  msg;                    /* Message to return            */

        mask = disable();
        prptr = &proctab[currpid];
        if (prptr->prhasmsg == FALSE) {
                prptr->prstate = PR_RECV;
                resched();              /* Block until message arrives  */
        }
        msg = prptr->prmsg;             /* Retrieve message             */
        prptr->prhasmsg = FALSE;        /* Reset message flag           */
        restore(mask);
        return msg;
}
```

Look carefully at the code and notice that *receive* copies the message from the process table entry into local variable *msg* and then returns the value in *msg*. Interestingly, *receive* does not modify field *prmsg* in the process table. Thus, it may seem that a more efficient implementation would avoid copying into a local variable and simply return the message from the process table:

```
return proctab[currpid].prmsg;
```

Unfortunately, such an implementation is incorrect; an exercise asks readers to consider why the implementation can produce incorrect results.

8.7 Implementation Of Non-Blocking Message Reception

Recvclr operates much like *receive* except that it always returns immediately. If a message is waiting, *recvclr* returns the message; otherwise, *recvclr* returns *OK*.

```
/* recvclr.c - recvclr */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  recvclr  -  Clear incoming message, and return message if one waiting
 *------------------------------------------------------------------------
 */
umsg32  recvclr(void)
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  procent *prptr;         /* Ptr to process' table entry  */
        umsg32  msg;                    /* Message to return            */

        mask = disable();
        prptr = &proctab[currpid];
        if (prptr->prhasmsg == TRUE) {
                msg = prptr->prmsg;     /* Retrieve message             */
                prptr->prhasmsg = FALSE;/* Reset message flag           */
        } else {
                msg = OK;
        }
        restore(mask);
        return msg;
}
```

8.8 Perspective

Like the counting semaphore abstraction in the previous chapter, the code for a basic message passing facility is extremely compact and efficient. Look at the functions and notice how few lines of code perform each operation. Furthermore, observe that storing the message buffer in the process table is important because doing so isolates message passing from memory management and allows message passing to be positioned at a low level in the hierarchy.

8.9 Summary

Message passing facilities provide inter-process communication that allows a process to send information to another process. A completely synchronous message passing system blocks either the sender or receiver, depending on how many messages have been sent and received. Our example system includes two facilities for message passing: a low-level mechanism that provides direct communication among processes and a high-level mechanism that uses rendezvous points.

The Xinu low-level message passing mechanism limits the message size to a single word, restricts each process to at most one outstanding message, and uses first-message semantics. Message storage is associated with the process table — a message sent to process *P* is stored in the process table entry for *P*. The use of first-message semantics allows a process to determine which of several events occurs first.

The low-level message facility comprises three functions: *send*, *receive*, and *recvclr*. Of the three functions, only *receive* is blocking — it blocks the calling process until a message arrives. A process can use *recvclr* to remove an old message before starting an interaction that uses message passing.

EXERCISES

8.1 Write a program that prints a prompt, and then loops printing the prompt again every 8 seconds until someone types a character. Hint: sleep(8) delays the calling process for 8 seconds.

8.2 Assume *send* and *receive* do not exist, and build code to perform message passing using *suspend* and *resume*.

8.3 The example implementation uses first-message semantics. What facilities exist to handle last-message semantics?

8.4 Implement versions of *send* and *receive* that record up to *K* messages per process (make successive calls to *send* block).

8.5 Investigate systems in which the innermost level of the system implements message passing instead of context switching. What is the advantage? The chief liability?

8.6 Consider the modification of *receive* mentioned in the text that returns the message directly from the process table entry:

```
return proctab[currpid].prmsg;
```

Explain why such an implementation is incorrect.

8.7 Implement versions of *send* and *receive* that define a fixed set of thirty-two possible messages. Instead of using integers to represent messages, use one bit of a word to represent each message, and allow a process to accumulate all thirty-two messages.

8.8 Observe that because *receive* uses *SYSERR* to indicate an error, sending a message with the same value as *SYSERR* is ambiguous. Furthermore, *recvclr* returns *OK* if no message is waiting. Modify *recvclr* to return *SYSERR* if no message is waiting, and modify *send* so it refuses to send *SYSERR* (i.e., checks its argument and returns an error if the value is *SYSERR*).

Chapter Contents

9.1 Introduction, 153
9.2 Types Of Memory, 153
9.3 Definition Of A Heavyweight Process, 154
9.4 Memory Management In Our Example System, 155
9.5 Program Segments And Regions Of Memory, 156
9.6 Dynamic Memory Allocation, 157
9.7 Design Of The Low–level Memory Manager, 158
9.8 Allocation Strategy And Memory Persistence, 159
9.9 Keeping Track Of Free Memory, 159
9.10 Implementation Of Low–level Memory Management, 160
9.11 Data Structure Definitions Used With Free Memory, 161
9.12 Allocating Heap Storage, 162
9.13 Allocating Stack Storage, 165
9.14 Releasing Heap And Stack Storage, 167
9.15 Perspective, 170
9.16 Summary, 170

9

Basic Memory Management

Memory is the ghost of experience.

— Anonymous

9.1 Introduction

Previous chapters explain concurrent computation and the facilities that an operating system provides to manage concurrent processes. The chapters discuss process creation and termination, scheduling, context switching, coordination, and inter-process communication.

This chapter begins the discussion of a second key topic: facilities that an operating system uses to manage memory. The chapter focuses on basics: dynamic allocation of stack and heap storage. It presents a set of functions that allocate and free memory, and explains how an embedded system handles memory for processes. The next chapter continues the discussion of memory management by describing address spaces, high-level memory management facilities, and virtual memory.

9.2 Types Of Memory

Because it is essential for program execution and data storage, main memory ranks high among the important resources that an operating system manages. An operating system maintains information about the size and location of available free memory blocks, and allocates memory to concurrent programs upon request. The system recovers the memory allocated to a process when the process terminates, making the memory available for reuse. We will assume that code remains resident in memory.

In many embedded systems, main memory consists of a contiguous set of locations with addresses *0* through *N–1*. However, some systems have complex address spaces where certain regions are reserved (e.g., for memory-mapped devices or backward compatibility). Our example platforms illustrate the two styles: the BeagleBone Black has a contiguous set of memory locations, and the Galileo provides a set of discontiguous blocks.

Broadly speaking, memory can be divided into two categories:

- *Stable Storage*, such as *Flash memory* that retains values after the power is removed
- *Random Access Memory* (*RAM*) that only retains values while the system is powered on

In terms of addresses, the two types of memory occupy separate locations. For example, addresses *0* through *K–1* might correspond to Flash memory, and addresses *K* through *N–1* might correspond to dynamic RAM.†

Some systems further distinguish regions of memory by using specific types of memory technologies. For example, RAM might be divided into two regions that use:

- *Static RAM* (*SRAM*): faster, but more expensive
- *Dynamic RAM* (*DRAM*): less expensive, but slower

Because SRAM is more expensive, systems usually have a small amount of SRAM and a larger amount of DRAM. If memory types differ, a programmer must carefully place variables and code that are referenced the most frequently in SRAM, and items that are referenced less often in DRAM.

9.3 Definition Of A Heavyweight Process

Operating systems include mechanisms that prevent an application from reading or modifying areas of memory that have been assigned to another application. For example, Chapter 10 discusses how each process can be assigned a separate virtual address space. The approach, which is known as a *heavyweight process abstraction*, creates an address space, and then creates a process to run in the address space. Usually, code for a heavyweight process is loaded dynamically — an application must be compiled and stored in a file on disk before the application can be used in a heavyweight process. Thus, when creating the heavyweight process, a programmer specifies the file on disk that contains the compiled code, and the operating system loads the specified application into a new virtual address space and starts a process executing the application.

Interestingly, some operating systems that support a heavyweight process abstraction also incorporate a *lightweight process abstraction* (i.e., *threads of execution*). Instead of a single process executing in an address space, the operating system starts a

†Usually, *K* and *N* are each a power of 2.

single process, but then permits the process to create additional threads that all execute in the same address space.

In the hybrid system described above, a thread is similar to a Xinu process. Each thread has a separate runtime stack used to hold activation records for function calls (including a copy of local variables). The stacks are allocated from the data area in the heavyweight process's address space. All the threads within a given heavyweight process share global variables; the global variables are allocated in the data area of the heavyweight process's address space. Sharing implies the need for coordination — threads within a heavyweight process must use synchronization primitives, such as semaphores, to control access to the variables they share. Figure 9.1 illustrates threads within a heavyweight process.

Figure 9.1 Illustration of a heavyweight process that contains multiple lightweight processes (threads). All threads share the address space.

9.4 Memory Management In Our Example System

The largest embedded systems, such as those used in video game consoles, have the memory management hardware and operating system support needed for demand paged virtual memory. On the smallest embedded systems, however, the hardware cannot support multiple address spaces, nor can it protect processes from one another. As a consequence, the operating system and all processes occupy a single address space.

Although running multiple processes in a single address space does not offer protection, the approach has advantages. Because they can pass pointers among themselves, processes can share large amounts of data without copying from one address space to another. Furthermore, the operating system itself can easily dereference an arbitrary pointer because the interpretation of an address does not depend on the process

context. Finally, having just one address space makes the memory manager much simpler than the memory managers found in more sophisticated systems.

9.5 Program Segments And Regions Of Memory

A C compiler divides a memory image into four contiguous regions known as:

- Text segment
- Data segment
- Bss segment
- Free space

Text segment. The text segment, which begins at the lowest usable memory address, contains compiled code for each of the functions that are part of the memory image (including the code for function *main*, which is compiled and linked like other functions). A compiler may also choose to place constants in the text segment because constants are read, but never changed. For example, string constants can be placed in the text segment. If the hardware includes a protection mechanism, addresses in the text segment are classified as *read-only*, which means that an error occurs if the program attempts to store into any of the text locations at runtime.

Data segment. The data segment, which follows the text segment, contains storage for all global variables that are assigned an initial value. In C, such variables are declared to be *global* by placing the declarations outside of function declarations. Values in the data segment are classified as *read-write* because they may be both accessed and modified.

Bss segment. The term *bss* abbreviates *block started by symbol*, and is taken from the assembly language of a PDP-11, the computer on which C was designed. The bss segment, which follows the data segment, contains global variables that are not initialized explicitly. Following C conventions, Xinu writes zero into each bss location before execution begins.

Free space. Addresses beyond the bss segment are considered *free* (i.e., unallocated) when execution begins. To simplify the discussion, we will assume free space consists of a single, contiguous region. In practice, some hardware platforms (e.g., the Galileo platform) may divide the free space into a set of discontiguous blocks.

As described in Chapter 3, a C program loader defines three external symbols, *etext*, *edata*, and *end*,† that correspond to the first memory location beyond the text segment, the first memory location beyond the data segment, and the first memory location beyond the bss segment. Figure 9.2 illustrates the memory layout when Xinu begins to execute and the three external symbols.

†External symbol names have an underscore prepended by the loader. Thus, if we use *etext* in a C program, it becomes *_etext* in the linked image.

Figure 9.2 Illustration of the memory layout when Xinu begins.

The external symbols shown in Figure 9.2 are not variables, but instead are names assigned to memory locations when the image is linked. Thus, a program should only use one of the external symbols to reference a location and should not load or store a value. For example, the text segment might occupy memory locations from the starting address through *etext–1*. To compute the size, a program declares *etext* to be an external integer, and references *&etext* in an expression.

How can a program determine the free space available for use? Our example platforms illustrate two approaches. In the case of the BeagleBone Black, the hardware manual specifies the available addresses. In the case of the Galileo, when it is powered on, the hardware and boot loader work together to probe memory, build a table of available address blocks, and pass the table to whatever operating system is being booted. We will see that the version of Xinu used on the BeagleBone Black contains constants that specify the address range, and the version of Xinu that runs on a Galileo is passed a list of available address blocks when it boots.

9.6 Dynamic Memory Allocation

Although they are allocated fixed locations in the address space and must remain resident in physical memory at all times, program text and global variables only account for part of the memory used by an executing process. The remaining memory can be allocated dynamically for:

- Stack
- Heap

Stack. Each process needs space for a *stack* that holds the activation record associated with each function the process invokes. In addition to arguments, an activation record contains storage for local variables.

Heap. A process or set of processes may also use *heap* storage. Items from the heap are allocated dynamically and persist independent of specific function calls.

Xinu accommodates both types of dynamic memory. First, when creating a new process, Xinu allocates a stack for the process. Stacks are allocated from the highest block in free space that accommodates the request. Second, whenever a process requests heap storage, Xinu allocates the necessary amount of space from the lowest free

block that accommodates the request. Figure 9.3 shows an example of the memory layout when three processes are executing and heap storage has been allocated.

Figure 9.3 Illustration of memory after three processes have been created.

9.7 Design Of The Low–level Memory Manager

A set of functions and associated data structures are used to manage free memory. The low-level memory manager provides five functions:

- *getstk* — Allocate stack space when a process is created
- *freestk* — Release a stack when a process terminates
- *getmem* — Allocate heap storage on demand
- *freemem* — Release heap storage as requested
- *meminit* — Initialize the free list at startup

Our design treats free space as a single, exhaustable resource — the low-level memory manager allocates space provided a request can be satisfied. Furthermore, the low-level memory manager does not partition the free space into memory available for process stacks and memory available for heap variables. Requests of one type can take the remaining free space, and leave none for the other type. Of course, such an allocation only works if processes cooperate. Otherwise, a given process can consume all free memory, leaving no space for other processes. Chapter 10 illustrates an alternative approach by describing a set of high-level memory management functions that prevent exhaustion by partitioning memory among subsystems. The high-level memory manager also demonstrates how processes can block until memory becomes available.

Functions *getstk* and *freestk* are not intended for general use. Instead, when it forms a new process, *create* calls *getstk* to allocate a stack. *Getstk* obtains a block of memory from the highest address of free space, and returns a pointer to the block. *Create* records the size and location of the allocated stack space in the process table entry. Later, when the process becomes current, the context switch moves the stack address into the stack pointer register, allowing the stack to be used for function calls. Finally, when the process terminates, *kill* calls function *freestk* to release the process's stack and return the block to the free list.

Functions *getmem* and *freemem* perform analogous functions for heap storage. Unlike the stack allocation functions, *getmem* and *freemem* allocate heap space from the lowest available free memory address.

9.8 Allocation Strategy And Memory Persistence

Because only *create* and *kill* allocate and free process stacks, the system can guarantee that the stack space allocated to a process will be released when the process exits. However, the system does not record the set of blocks that a process allocates from the heap by calling *getmem*. Therefore, the system does not automatically release heap storage. As a consequence, the burden of returning heap space is left to the programmer:

> *Heap space persists independent of the process that allocates the space. Before it exits, a process must explicitly release storage that it has allocated from the heap, or the space will remain allocated.*

Of course, returning allocated heap space does not guarantee that the heap will never be exhausted. On the one hand, demand can exceed the available space. On the other hand, the free space can become *fragmented* into small, discontiguous pieces that are each too small to satisfy a request. A later chapter continues the discussion of allocation policies, and shows one approach to avoiding fragmentation of the free space.

9.9 Keeping Track Of Free Memory

A memory manager must keep information about all free memory blocks. To do so, the memory manager forms a list, where each item on the list specifies a memory address at which a block starts and a length. In systems where the entire free memory is contiguous, the initial list will contain only one item that corresponds to the block of memory between the end of the program and the highest available address. In systems where the free memory is divided into blocks, the initial list will contain one node for each block. In either case, when a process requests a block of memory, the memory manager searches the list, finds a free area, allocates the requested size block, and updates the list to show that more of the free memory has been allocated. Similarly, whenever a process releases a previously allocated block of memory, the memory manager adds the block to the list of free blocks. Figure 9.4 illustrates an example set of four free memory blocks.

Block	Address	Length
1	0x84F800	4096
2	0x850F70	8192
3	0x8A03F0	8192
4	0x8C01D0	4096

Figure 9.4 An example set of free memory blocks.

A memory manager must examine each transaction carefully to avoid generating an arbitrarily long list of small blocks. When a block is released, the memory manager scans the list to see if the released block is adjacent to the end of one of the existing free blocks. If so, the size of the existing block can be increased without adding a new entry to the list. Similarly, if the new block is adjacent to the beginning of an existing block, the entry can be updated. Finally, if a released block exactly fills the gap between two free blocks on the list, the memory manager will combine the two existing entries on the list into one giant block that covers all the memory from the two on the list plus the released block. We use the term *coalesce* to describe combining entries. The point is: if a memory manager is built correctly, once all allocated blocks have been released, the free list will be back to the initial state (i.e., the exact set of memory regions that were free when the operating system started).

9.10 Implementation Of Low–level Memory Management

Where should the list of free memory blocks be stored? Our example implementation follows a standard approach by using free memory itself to store the list. After all, a free memory block is not being used, so the contents are no longer needed. Thus, the free memory blocks can be chained together to form a linked list by placing a pointer in each block to the next block.

In the code, global variable *memlist* contains a pointer to the first free block. The key to understanding the implementation lies in the invariant maintained at all times:

> *All free blocks of memory are kept on a linked list; blocks on the free list are ordered by increasing address.*

The example set of blocks in Figure 9.4 shows two items associated with each entry: an address and a size. In our linked list implementation, each node in the list points to (i.e., gives the address) of the next node. However, we must also store the size of each block. Therefore, each block of free memory contains two items: a pointer to the next free block of memory and an integer that gives the size of the current block. Figure 9.5 illustrates the concept.

Figure 9.5 Illustration of a free memory list that contains two memory blocks.

9.11 Data Structure Definitions Used With Free Memory

File *memory.h* contains declarations related to memory management. Structure *memblk* is imposed on each free node.

```
/* memory.h - roundmb, truncmb, freestk */

#define PAGE_SIZE       4096

/*------------------------------------------------------------------------
 * roundmb, truncmb - Round or truncate address to memory block size
 *------------------------------------------------------------------------
 */
#define roundmb(x)      (char *)( (7 + (uint32)(x)) & (~7) )
#define truncmb(x)      (char *)( ((uint32)(x)) & (~7) )

/*------------------------------------------------------------------------
 *  freestk  --  Free stack memory allocated by getstk
 *------------------------------------------------------------------------
 */
#define freestk(p,len)  freemem((char *)((uint32)(p)            \
                                - ((uint32)roundmb(len))        \
                                + (uint32)sizeof(uint32)),      \
                                (uint32)roundmb(len) )

struct  memblk  {                       /* See roundmb & truncmb        */
        struct  memblk  *mnext;         /* Ptr to next free memory blk  */
        uint32  mlength;                /* Size of blk (includes memblk)*/
        };
extern  struct  memblk  memlist;        /* Head of free memory list     */
extern  void    *minheap;               /* Start of heap                */
extern  void    *maxheap;               /* Highest valid heap address   */


/* Added by linker */

extern  int     text;                   /* Start of text segment        */
extern  int     etext;                  /* End of text segment          */
extern  int     data;                   /* Start of data segment        */
extern  int     edata;                  /* End of data segment          */
extern  int     bss;                    /* Start of bss segment         */
extern  int     ebss;                   /* End of bss segment           */
extern  int     end;                    /* End of program               */
```

In struct *memblk*, field *mnext* points to the next block on the list or contains the value *NULL* to indicate that a block is the final block on the list. Field *mlength* specifies the length of the current block in bytes, including the header. Note that the length is declared to be an unsigned long, which accommodates any size block up to the entire 32-bit physical address space.

Variable *memlist*, which constitutes the head of the list, is defined as consisting of a *memblk* structure. Thus, the head of the list has exactly the same form that we impose on other nodes in the list. In *memlist*, however, the *mlength* field (which is normally used to store the size of a block) is not meaningful because the size of *memlist* is *sizeof(struct memblk)*. Consequently, we can use the length field for another purpose. Xinu uses the field to store the total size of free memory (i.e., a sum of the length field in each free block). Having a count of the free memory can help when debugging or to assess whether the system is approaching the maximum possible size.

Note that each block on the free list must hold a complete *memblk* structure (i.e., eight bytes). The design choice has a consequence: our free list cannot store a free block of less then eight bytes. How can we guarantee that no process attempts to free a smaller amount of memory? We can tell programmers that they must free exactly the same amount they request, and design the memory management functions to ensure all requests are at least eight bytes. But another problem can arise if the memory manager extracts a piece from a free block: subtraction can leave a remainder of less than eight bytes. To solve the problem, our memory manager rounds all requests to a multiple of *memblk* structures. Our solution has a useful property described later: subtracting a multiple of eight from a multiple of eight always produces a multiple of eight.

File *memory.h* defines two inline functions, *roundmb* and *truncmb*, that handle the rounding. Function *roundmb* rounds requests to multiples of eight bytes, and *truncmb* is used to truncate a memory size to a multiple of eight bytes. Truncation is only used at startup: if the initial size of a free block is not a multiple of eight, the size must be truncated down rather than rounded up. To make the implementation efficient, the code for the two functions uses constants and Boolean operations rather than the *sizeof* function and division. Using Boolean operations is only possible because the size of a memory block is a power of two. We can summarize:

> *Rounding all requests to multiples of the* memblk *structure ensures that each request satisfies the size constraint and guarantees that no free block will ever be too small to link into the free list.*

9.12 Allocating Heap Storage

Function *getmem* allocates heap storage by finding a free block that is sufficient for the request. Our implementation uses a *first-fit* allocation strategy by allocating the first block on the free list that satisfies a request. *Getmem* subtracts the requested memory from the free block and adjusts the free list accordingly. File *getmem.c* contains the code.

```
/* getmem.c - getmem */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  getmem  -  Allocate heap storage, returning lowest word address
 *------------------------------------------------------------------------
 */
char    *getmem(
          uint32        nbytes          /* Size of memory requested     */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  memblk  *prev, *curr, *leftover;

        mask = disable();
        if (nbytes == 0) {
                restore(mask);
                return (char *)SYSERR;
        }

        nbytes = (uint32) roundmb(nbytes);      /* Use memblk multiples */

        prev = &memlist;
        curr = memlist.mnext;
        while (curr != NULL) {                  /* Search free list     */

                if (curr->mlength == nbytes) {  /* Block is exact match */
                        prev->mnext = curr->mnext;
                        memlist.mlength -= nbytes;
                        restore(mask);
                        return (char *)(curr);

                } else if (curr->mlength > nbytes) { /* Split big block */
                        leftover = (struct memblk *)((uint32) curr +
                                        nbytes);
                        prev->mnext = leftover;
                        leftover->mnext = curr->mnext;
                        leftover->mlength = curr->mlength - nbytes;
                        memlist.mlength -= nbytes;
                        restore(mask);
                        return (char *)(curr);
                } else {                        /* Move to next block   */
                        prev = curr;
                        curr = curr->mnext;
```

```
            }
        }
        restore(mask);
        return (char *)SYSERR;
}
```

After verifying that its argument is valid and the free list is not empty, *getmem* uses *roundmb* to round the memory request to a multiple of *memblk* bytes, and then searches the free list to find the first block of memory large enough to satisfy the request. Because the free list is singly linked, *getmem* uses two pointers, *prev* and *curr*, to walk the list. The code maintains the following invariant during the search: when *curr* points to a free block, *prev* points to its predecessor on the list (possibly the head of the list, *memlist*). As it runs through the list, the code must ensure the invariant remains intact. Consequently, when a free block is discovered that is large enough to satisfy the request, *prev* will point to the predecessor.

At each step of the iteration, *getmem* compares the size of the current block to *nbytes*, the size of the request. There are three cases. If the size of the current block is less than the size requested, *getmem* moves to the next block on the list and continues the search. If the size of the current block exactly matches the size of the request, *getmem* removes the block from the free list (by making the *mnext* field in the predecessor block point to the successor block), and returns a pointer to the current block. Finally, if the size of the current block is greater than the size requested, *getmem* partitions the current block into two pieces: one of size *nbytes* that will be returned to the caller, and a remaining piece that will be left on the free list. To divide a free block into two pieces, *getmem* computes the address of the remaining piece, and places the address in variable *leftover*. Computing such an address is conceptually simple: the leftover piece lies *nbytes* beyond the beginning of the block. However, adding *nbytes* to pointer *curr* will not produce the desired result because C performs pointer arithmetic. To force C to use integer arithmetic instead of pointer arithmetic, variable *curr* is cast to an unsigned integer before adding *nbytes*:

```
(uint32)curr
```

Once a sum has been computed, another cast is applied to change the result back into a memory block pointer. After *leftover* has been computed, the *mnext* field of the *prev* block is updated, and the *mnext* and *mlength* fields in the *leftover* block are assigned.

The code relies on a fundamental mathematical relationship: subtracting two multiples of K will produce a multiple of K. In the example, K is the size of a *memblk* structure, eight bytes. Thus, if the system begins by using *roundmb* to round the size of free memory, and always uses *roundmb* to round requests, each free block and each leftover piece will be large enough to hold a *memblk* structure.

9.13 Allocating Stack Storage

Function *getstk* allocates a block of memory for a process stack. We will see that *getstk* is invoked whenever a process is created. The code appears in file *getstk.c*.

Because the free list is kept in order by memory address and stack space is allocated from the highest available block, *getstk* must search the entire list of free blocks. During the search, *getstk* records the address of any block that satisfies the request, which means that after the search completes, the last recorded address points to the free block with the highest address that satisfies the request.† As with *getmem*, *getstk* maintains the invariant that during the search, variables *curr* and *prev* point to a free block of memory and the predecessor of the free block, respectively. Whenever a block is found that has sufficient size to satisfy the request, *getstk* sets variable *fits* to the address of the block and sets variable *fitsprev* to the address of its predecessor. Thus, when the search completes, *fits* points to the usable free block with the highest memory address (or will remain equal to *NULL* if no block satisfies the request).

Once the search completes and a block has been found, two cases arise, analogous to the cases in *getmem*. If the size of the block on the free list is exactly the size requested, *getstk* unlinks the block from the free list and returns the address of the block to its caller. Otherwise, *getstk* partitions the block into two pieces, allocating a piece of size *nbytes*, and leaves the remainder on the free list. Because *getstk* returns the piece from the highest part of the selected block, the computation differs slightly from that in *getmem*.

†The strategy of allocating the block with the highest address that satisfies a request is known as the *last-fit* strategy.

```
/* getstk.c - getstk */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  getstk  -  Allocate stack memory, returning highest word address
 *------------------------------------------------------------------------
 */
char    *getstk(
          uint32        nbytes          /* Size of memory requested     */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  memblk  *prev, *curr;   /* Walk through memory list     */
        struct  memblk  *fits, *fitsprev; /* Record block that fits     */

        mask = disable();
        if (nbytes == 0) {
                restore(mask);
                return (char *)SYSERR;
        }

        nbytes = (uint32) roundmb(nbytes);      /* Use mblock multiples */

        prev = &memlist;
        curr = memlist.mnext;
        fits = NULL;
        fitsprev = NULL;  /* Just to avoid a compiler warning */

        while (curr != NULL) {                  /* Scan entire list     */
                if (curr->mlength >= nbytes) {  /* Record block address */
                        fits = curr;            /*   when request fits  */
                        fitsprev = prev;
                }
                prev = curr;
                curr = curr->mnext;
        }

        if (fits == NULL) {                     /* No block was found   */
                restore(mask);
                return (char *)SYSERR;
        }
        if (nbytes == fits->mlength) {          /* Block is exact match */
                fitsprev->mnext = fits->mnext;
        } else {                                /* Remove top section   */
```

```
                fits->mlength -= nbytes;
                fits = (struct memblk *)((uint32)fits + fits->mlength);
        }
        memlist.mlength -= nbytes;
        restore(mask);
        return (char *)((uint32) fits + nbytes - sizeof(uint32));
}
```

9.14 Releasing Heap And Stack Storage

Once it finishes using a block of heap storage, a process calls function *freemem* to return the block to the free list, making the memory eligible for subsequent allocation. Because blocks on the free list are kept in order by address, *freemem* uses the block's address to find the correct location on the list. In addition, *freemem* handles the task of *coalescing* the block with adjacent free blocks. There are three cases: the new block of memory can be adjacent to the previous block, adjacent to the succeeding block, or adjacent to both. When any of the three cases occurs, *freemem* combines the new block with adjacent block(s) to form one large block on the free list. Coalescing helps avoid memory fragmentation.

The code for *freemem* can be found in file *freemem.c*. As in *getmem*, two pointers, *prev* and *next*, run through the list of free blocks. *Freemem* searches the list until the address of the block to be returned lies between *prev* and *next*. Once the correct position has been found, the code performs coalescing.

Coalescing is handled in three steps. The code first checks for coalescing with the previous block. That is, *freemem* adds the length of the previous block to the block's address to compute the address one beyond the previous block. *Freemem* compares the result, which is found in variable *top*, to the address of the block being inserted. If the address of the inserted block equals *top*, *freemem* increases the size of the previous block to include the new block. Otherwise, freemem inserts the new block in the list. Of course, if the previous pointer points to the head of the *memlist*, no coalescing can be performed.

Once it has handled coalescing with the previous block, *freemem* checks for coalescing with the next block. Once again, *freemem* computes the address that lies one beyond the current block, and tests whether the address is equal to the address of the next block. If so, the current block is adjacent to the next block, so *freemem* increases the size of the current block to include the next block, and unlinks the next block from the list.

The important point is that *freemem* handles all three special cases:

> *When adding a block to the free list, the memory manager must check to see whether the new block is adjacent to the previous block, adjacent to the next block, or adjacent to both.*

```
/* freemem.c - freemem */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  freemem  -  Free a memory block, returning the block to the free list
 *------------------------------------------------------------------------
 */
syscall freemem(
          char          *blkaddr,       /* Pointer to memory block      */
          uint32        nbytes          /* Size of block in bytes       */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  memblk  *next, *prev, *block;
        uint32  top;

        mask = disable();
        if ((nbytes == 0) || ((uint32) blkaddr < (uint32) minheap)
                          || ((uint32) blkaddr > (uint32) maxheap)) {
                restore(mask);
                return SYSERR;
        }

        nbytes = (uint32) roundmb(nbytes);      /* Use memblk multiples */
        block = (struct memblk *)blkaddr;

        prev = &memlist;                        /* Walk along free list */
        next = memlist.mnext;
        while ((next != NULL) && (next < block)) {
                prev = next;
                next = next->mnext;
        }

        if (prev == &memlist) {         /* Compute top of previous block*/
                top = (uint32) NULL;
        } else {
                top = (uint32) prev + prev->mlength;
        }

        /* Ensure new block does not overlap previous or next blocks    */

        if (((prev != &memlist) && (uint32) block < top)
            || ((next != NULL)  && (uint32) block+nbytes>(uint32)next)) {
                restore(mask);
```

```
                return SYSERR;
        }

        memlist.mlength += nbytes;

        /* Either coalesce with previous block or add to free list */

        if (top == (uint32) block) {    /* Coalesce with previous block */
                prev->mlength += nbytes;
                block = prev;
        } else {                        /* Link into list as new node   */
                block->mnext = next;
                block->mlength = nbytes;
                prev->mnext = block;
        }

        /* Coalesce with next block if adjacent */

        if (((uint32) block + block->mlength) == (uint32) next) {
                block->mlength += next->mlength;
                block->mnext = next->mnext;
        }
        restore(mask);
        return OK;
}
```

Because free memory is treated as a single resource that can be used for stacks or heap storage, releasing stack memory follows the same algorithm as releasing heap storage. The only difference between heap and stack allocations arises because *getmem* returns the lowest address of an allocated block and *getstk* returns the highest address. In the current implementation, *freestk* is an inline function that invokes *freemem*. Before calling *freemem*, *freestk* must convert its argument from the highest address in a block to the lowest. The code is found in memory.h.† Although the current implementation uses a single underlying list, keeping *freestk* separate from *freemem* maintains a conceptual distinction, and makes it easier to modify the implementation later. The point is:

> *Although the current implementation uses the same underlying function to release heap and stack storage, having separate system calls for* freestk *and* freemem *maintains the conceptual distinction and makes the system easier to change later.*

†File memory.h can be found on page 161.

9.15 Perspective

Although the mechanisms are relatively straightforward, the design of a memory management subsystem reveals one of the most surprisingly subtle problems in operating systems. The problem arises from a fundamental conflict. On the one hand, an operating system is designed to run without stopping. Therefore, operating system facilities must be resource-preserving: when a process finishes with a resource, the system must recapture the resource and make it available to other processes. On the other hand, any memory management mechanism that allows processes to allocate and free blocks of arbitrary size is not resource-preserving because memory can become fragmented, with free memory divided into small, discontiguous blocks. Thus, a designer must be aware that the choice represents a tradeoff. Allowing arbitrary-size allocations makes the system easier to use, but also introduces a potential for problems.

9.16 Summary

We use the term *heavyweight process* to refer to an application that runs in a separate address space; a lightweight process abstraction permits one or more processes to run in each address space.

A Xinu image contains three segments: a text segment consisting of compiled code, a data segment that contains initialized data values, and a bss segment that contains uninitialized variables. When a system starts, physical memory not allocated to the three segments is considered free, and a low-level memory manager allocates the free memory on demand.

When Xinu starts, the low-level memory manager forms a linked list of all free memory blocks; both stack and heap storage is allocated from the list as needed. Heap storage is allocated by finding the first free memory block that satisfies the request (i.e., the free block with the lowest address). A request for stack storage is satisfied from the highest free memory block that satisfies the request. Because the list of free memory blocks is singly linked in order by address, allocating stack space requires searching the entire free list.

Our low-level memory manager treats free space as an exhaustable resource with no partition between stack and heap storage. Because the memory manager does not contain mechanisms to prevent a process from allocating all free memory, a programmer must plan carefully to avoid starvation.

EXERCISES

9.1 Write a function that walks the list of free memory blocks and prints a line with the address and length of each block.

9.2 In the previous exercise, what can occur if your function does not disable interrupts while searching the list?

9.3 An early version of the low-level memory manager had no provision for returning blocks of memory to the free list. Speculate about memory allocation in an embedded system: are *freemem* and *freestk* necessary? Why or why not?

9.4 Replace the low-level memory management functions with a set of functions that allocate heap and stack memory permanently (i.e., without providing a mechanism to return storage to a free list). How do the sizes of the new allocation routines compare to the sizes of *getstk* and *getmem*?

9.5 Does the approach of allocating stack and heap storage from opposite ends of free space help minimize fragmentation? To find out, consider a series of requests that intermix allocation and freeing of stack storage of 1000 bytes and heap storage of 500 bytes. Compare the approach described in the chapter to an approach that allocates stack and heap requests from the same end of free space (i.e., an approach in which all allocation uses *getmem*). Find a sequence of requests that result in fragmentation if stack and heap requests are not allocated from separate ends.

9.6 At startup, function *meminit* is called to form an initial list of free memory. Compare *meminit* for the Galileo and BeagleBone Black platforms. Which is smaller? Which is more general? Explain the basic tradeoff.

9.7 Many embedded systems go through a prototype stage, in which the system is built on a general platform, and a final stage, in which minimal hardware is designed for the system. In terms of memory management, one question concerns the size of the stack needed by each process. Modify the code to allow the system to measure the maximum stack space used by a process and report the maximum stack size when the process exits.

9.8 Consider a hardware platform where the operating system must probe to find the size of memory at startup (i.e., read from an address and either receive an exception or some random value). Describe a strategy to find the maximum valid address efficiently.

Chapter Contents

10.1 Introduction, 175
10.2 Partitioned Space Allocation, 176
10.3 Buffer Pools, 176
10.4 Allocating A Buffer, 178
10.5 Returning Buffers To The Buffer Pool, 179
10.6 Creating A Buffer Pool, 181
10.7 Initializing The Buffer Pool Table, 183
10.8 Virtual Memory And Memory Multiplexing, 184
10.9 Real And Virtual Address Spaces, 185
10.10 Hardware For Demand Paging, 186
10.11 Address Translation With A Page Table, 187
10.12 Metadata In A Page Table Entry, 188
10.13 Demand Paging And Design Questions, 189
10.14 Page Replacement And Global Clock, 190
10.15 Perspective, 191
10.16 Summary, 191

10

High-level Memory Management and Virtual Memory

Yea, from the table of my memory I'll wipe away all fond trivial records.

— William Shakespeare

10.1 Introduction

Earlier chapters consider abstractions that an operating system uses to manage computation and coordinate concurrent processes. The previous chapter describes a low-level memory management facility that treats memory as an exhaustable resource. The chapter discusses address spaces, program segments, and functions that manage a global free list. Although they are necessary, low-level memory management facilities are not sufficient for all needs.

This chapter completes the discussion of memory management by introducing high-level facilities. The chapter explains the motivation for partitioning memory resources into independent subsets. It presents a high-level memory management mechanism that allows memory to be divided into independent buffer pools, and explains how allocation and use of the memory in a given pool does not affect the use of memory in other pools. The chapter also describes virtual memory, and explains how virtual memory hardware operates.

10.2 Partitioned Space Allocation

Functions *getmem* and *freemem* that were described in the previous chapter constitute a basic memory manager. The design places no limit on the amount of memory that a given process can allocate, nor do the functions attempt to divide free space "fairly." Instead, the allocation functions merely honor requests on a first-come-first-served basis until no free memory remains. Once free memory has been exhausted, the functions reject further requests without blocking processes or waiting for memory to be released. Although it is relatively efficient, a global allocation strategy that forces all processes to contend for the same memory can lead to deprivation, a situation in which one or more processes cannot obtain memory because all memory has been consumed. As a consequence, global memory allocation schemes do not work well for all parts of the operating system.

To understand why a system cannot rely on global allocation, consider software for network communication. Packets arrive at random. Because a network application takes time to process a given packet, additional packets may arrive while one is being handled. If each incoming packet is placed in a memory buffer, exhaustive allocation can lead to disaster because incoming packets will pile up occupying memory and waiting to be processed. In the worst case, all the available memory will be allocated to packet buffers, and none will be available for other operating system functions. In particular, if disk I/O uses memory, all disk I/O may stop until memory becomes available. If the application processing network packets attempts to write to a file, *deadlock* can occur: the network process can block waiting for a disk buffer, but all memory is used for network buffers and no network buffer can be released until disk I/O completes.

To prevent deadlocks, higher-level memory management must be designed to partition free memory into independent subsets, and ensure that allocation and deallocation of a given subset remains independent from the allocation and deallocation of other subsets. By limiting the amount of memory that can be used for a particular function, the system can guarantee that excessive requests will not lead to global deprivation. Furthermore, the system can assume that memory allocated for a particular function will always be returned, so it can arrange to suspend processes until their memory request can be satisfied, eliminating the overhead introduced by busy waiting. Partitioning cannot guarantee that no deadlocks will occur, but it does limit unintentional deadlocks that arise when one subsystem takes memory needed by another subsystem.

10.3 Buffer Pools

The mechanism we have chosen to handle partitioned memory is known as a *buffer pool* manager. Memory is divided into a set of buffer pools. Each buffer pool contains a fixed number of memory blocks, and all blocks in a given pool are the same size. The term *buffer* was chosen to reflect the intended use in I/O routines and communication software (e.g., disk buffers or buffers for network packets).

The memory space for a particular set of buffers is allocated when the pool is created; once a pool has been allocated, there is no way to increase the number of buffers in the pool or to change the buffer size.

Each buffer pool is identified by an integer, known as a *pool identifier* or *buffer pool ID*. Like other IDs in Xinu, a buffer pool ID is used as an index into the buffer pool table, *buftab*. Once a pool has been created, a process uses the pool ID whenever it requests a buffer from a pool or releases a previously allocated buffer back to a pool. Requests to allocate or release a buffer from a pool do not need to specify the length of a buffer because the size of buffers is fixed when the pool is created.

The data structure used to hold information about buffer pools consists of a single table. Each entry in the table holds a buffer size, a semaphore ID, and a pointer to a linked list of buffers for the pool. Pertinent declarations can be found in file *bufpool.h*:

```
/* bufpool.h */

#ifndef NBPOOLS
#define NBPOOLS 20              /* Maximum number of buffer pools       */
#endif

#ifndef BP_MAXB
#define BP_MAXB 8192            /* Maximum buffer size in bytes         */
#endif

#define BP_MINB 8               /* Minimum buffer size in bytes         */
#ifndef BP_MAXN
#define BP_MAXN 2048            /* Maximum number of buffers in a pool  */
#endif

struct  bpentry {               /* Description of a single buffer pool  */
        struct  bpentry *bpnext;/* pointer to next free buffer          */
        sid32   bpsem;          /* semaphore that counts buffers        */
                                /*    currently available in the pool   */
        uint32  bpsize;         /* size of buffers in this pool         */
        };

extern  struct  bpentry buftab[];/* Buffer pool table                   */
extern  bpid32  nbpools;        /* current number of allocated pools    */
```

Structure *bpentry* defines the contents of an entry in the buffer pool table, *buftab*. The buffers for a given pool are linked into a list, with field *bpnext* pointing to the first buffer on the list. Semaphore *bpsem* controls allocation from the pool, and integer *bpsize* gives the length of buffers in the pool.

10.4 Allocating A Buffer

The buffer pool mechanism differs from the low-level memory manager in another way: the mechanism is *synchronous*. That is, a process that requests a buffer will be blocked until the request can be satisfied. As in many of the previous examples, the implementation uses semaphores to control access to a buffer pool. Each buffer pool has a semaphore, and the code that allocates a buffer calls *wait* on a pool's semaphore. The call returns immediately if buffers remain in the pool, but blocks the caller if no buffers remain. Eventually, when another process returns a buffer to a pool, the semaphore is signaled, which allows a waiting process to obtain the buffer and resume execution.

Three functions provide an interface to buffer pools. A process calls function *mkpool* to create a buffer pool and obtain an ID. Once a pool as been created, a process can call function *getbuf* to obtain a buffer, and function *freebuf* to release a buffer back to the pool. *Getbuf* works as expected, waiting on the semaphore until a buffer is available, and then unlinking the first buffer from the list. The code is found in file *getbuf.c*:

```
/* getbuf.c - getbuf */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  getbuf  -  Get a buffer from a preestablished buffer pool
 *------------------------------------------------------------------------
 */
char    *getbuf(
          bpid32        poolid          /* Index of pool in buftab      */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  bpentry *bpptr;         /* Pointer to entry in buftab   */
        struct  bpentry *bufptr;        /* Pointer to a buffer          */

        mask = disable();

        /* Check arguments */

        if ( (poolid < 0  ||  poolid >= nbpools) ) {
                restore(mask);
                return (char *)SYSERR;

        }
        bpptr = &buftab[poolid];
```

```
        /* Wait for pool to have > 0 buffers and allocate a buffer */

        wait(bpptr->bpsem);
        bufptr = bpptr->bpnext;

        /* Unlink buffer from pool */

        bpptr->bpnext = bufptr->bpnext;

        /* Record pool ID in first four bytes of buffer and skip */

        *(bpid32 *)bufptr = poolid;
        bufptr = (struct bpentry *)(sizeof(bpid32) + (char *)bufptr);
        restore(mask);
        return (char *)bufptr;
}
```

Observant readers may have noticed that *getbuf* does not return the address of the buffer to its caller. Instead, *getbuf* stores the pool ID in the first four bytes of the allocated space, and returns the address just beyond the ID. From a caller's point of view, a call to *getbuf* returns the address of a buffer; the caller does not need to worry that earlier bytes hold the pool ID. The system is transparent: when the pool is created, extra space is allocated in each buffer to hold the pool ID. When a buffer is released, *freebuf* uses the hidden pool ID to determine the pool to which a buffer belongs. The technique of using hidden information to identify a buffer pool turns out to be especially useful when buffers are returned by a process other than the one that allocated the buffer.

10.5 Returning Buffers To The Buffer Pool

Function *freebuf* returns a buffer to the pool from which it was allocated. The code is found in file *freebuf.c*:

```
/* freebuf.c - freebuf */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  freebuf  -  Free a buffer that was allocated from a pool by getbuf
 *------------------------------------------------------------------------
 */
syscall freebuf(
          char          *bufaddr        /* Address of buffer to return  */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  bpentry *bpptr;         /* Pointer to entry in buftab   */
        bpid32  poolid;                 /* ID of buffer's pool          */

        mask = disable();

        /* Extract pool ID from integer prior to buffer address */

        bufaddr -= sizeof(bpid32);
        poolid = *(bpid32 *)bufaddr;
        if (poolid < 0  ||  poolid >= nbpools) {
                restore(mask);
                return SYSERR;
        }

        /* Get address of correct pool entry in table */

        bpptr = &buftab[poolid];

        /* Insert buffer into list and signal semaphore */

        ((struct bpentry *)bufaddr)->bpnext = bpptr->bpnext;
        bpptr->bpnext = (struct bpentry *)bufaddr;
        signal(bpptr->bpsem);
        restore(mask);
        return OK;
}
```

Recall that when it allocates a buffer, *getbuf* stores the pool ID in the four bytes that precede the buffer address. *Freebuf* moves back four bytes from the beginning of the buffer, and extracts the pool ID. After verifying that the pool ID is valid, *getbuf* uses the ID to locate the entry in the table of buffer pools. It then links the buffer back

into the linked list of buffers, and signals the pool semaphore *bpsem*, allowing other processes to use the buffer.

10.6 Creating A Buffer Pool

Function *mkbufpool* creates a new buffer pool and returns its ID. *Mkbufpool* takes two arguments: the size of buffers in the pool and the number of buffers. *Mkbufpool* begins by checking its arguments. If the buffer size is out of range, the requested number of buffers is negative, or the buffer pool table is full, *mkbufpool* reports an error. *Mkbufpool* computes the size of memory required to hold the buffers, and calls *getmem* to allocate the needed memory. If the memory allocation succeeds, *mkbufpool* allocates an entry in the buffer pool table, and fills in entries. It creates a semaphore, saves the buffer size, and stores the address of the allocated memory in the linked list pointer, *bpnext*.

Once the table entry has been initialized, *mkbufpool* iterates through the allocated memory, dividing the block into a set of buffers. It links each buffer onto the free list. Note that when *mkbufpool* creates the free list, each block of memory contains the size of the buffer the user requested plus the size of a buffer pool ID (four bytes). Thus, after the buffer pool ID is stored in the block, sufficient bytes remain for the buffer that the user requested. After the free list has been formed, *mkbufpool* returns the pool ID to its caller.

```
/* mkbufpool.c - mkbufpool */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  mkbufpool  -  Allocate memory for a buffer pool and link the buffers
 *------------------------------------------------------------------------
 */
bpid32  mkbufpool(
          int32         bufsiz,         /* Size of a buffer in the pool */
          int32         numbufs         /* Number of buffers in the pool*/
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        bpid32  poolid;                 /* ID of pool that is created   */
        struct  bpentry *bpptr;         /* Pointer to entry in buftab   */
        char    *buf;                   /* Pointer to memory for buffer */

        mask = disable();
        if (bufsiz<BP_MINB || bufsiz>BP_MAXB
            || numbufs<1 || numbufs>BP_MAXN
            || nbpools >= NBPOOLS) {
                restore(mask);
                return (bpid32)SYSERR;
        }
        /* Round request to a multiple of 4 bytes */

        bufsiz = ( (bufsiz + 3) & (~3) );

        buf = (char *)getmem( numbufs * (bufsiz+sizeof(bpid32)) );
        if ((int32)buf == SYSERR) {
                restore(mask);
                return (bpid32)SYSERR;
        }
        poolid = nbpools++;
        bpptr = &buftab[poolid];
        bpptr->bpnext = (struct bpentry *)buf;
        bpptr->bpsize = bufsiz;
        if ( (bpptr->bpsem = semcreate(numbufs)) == SYSERR) {
                nbpools--;
                restore(mask);
                return (bpid32)SYSERR;
        }
        bufsiz+=sizeof(bpid32);
        for (numbufs-- ; numbufs>0 ; numbufs-- ) {
```

```
                bpptr = (struct bpentry *)buf;
                buf += bufsiz;
                bpptr->bpnext = (struct bpentry *)buf;
        }
        bpptr = (struct bpentry *)buf;
        bpptr->bpnext = (struct bpentry *)NULL;
        restore(mask);
        return poolid;
}
```

10.7 Initializing The Buffer Pool Table

Function *bufinit* initializes the buffer pool table. Initialization occurs once before the buffer pools are used (i.e., the table is initialized when the system boots). The code, found in file *bufinit.c*, is trivial:

```
/* bufinit.c - bufinit */

#include <xinu.h>

struct  bpentry buftab[NBPOOLS];                /* Buffer pool table    */
bpid32  nbpools;

/*------------------------------------------------------------------------
 *  bufinit  -  Initialize the buffer pool data structure
 *------------------------------------------------------------------------
 */
status  bufinit(void)
{
        nbpools = 0;
        return OK;
}
```

All that *bufinit* needs to do is set the global count of allocated buffer pools. In the example code, buffer pools can be allocated dynamically. However, once it has been allocated, a pool cannot be deallocated. An exercise suggests extending the mechanism to permit dynamic deallocation.

10.8 Virtual Memory And Memory Multiplexing

Most large computer systems virtualize memory and present an application with an abstract view of memory. Each application appears to occupy a large address space; on a system with a small physical memory, the address space can exceed the size of physical memory. The operating system multiplexes physical memory among all processes that need to use it, moving all or part of the application into physical memory as needed. That is, code and data for processes are kept on secondary storage (i.e., disk), and moved into main memory temporarily when the process is executing. Although few embedded systems rely on virtual memory, many processors include virtual memory hardware.

The chief design question in virtual memory management systems concerns the form of multiplexing. Several possibilities have been used:

- Swapping
- Segmentation
- Paging

Swapping refers to an approach that moves all code and data associated with a particular computation into main memory when the scheduler makes the computation current. Swapping works best for a long-running computation, such as a word processor that runs while a human types a document — the computation is moved into main memory, and remains resident for a long period.

Segmentation refers to an approach that moves pieces of the code and data associated with a computation into main memory as needed. One can imagine, for example, placing each function and the associated variables in a separate segment. When a function is called, the operating system loads the segment containing the function into main memory. Seldom-used functions (e.g., a function that displays an error message) remain on secondary storage. In theory, segmentation uses less memory than swapping because segmentation allows pieces of a computation to be loaded into memory as needed. Although the approach has intuitive appeal, few operating systems use dynamic segmentation.

Paging refers to an approach that divides each program into small, fixed-size pieces called *pages*. The system keeps the most recently referenced pages in main memory, and moves copies of other pages to secondary storage. Pages are fetched on demand — when a running program references memory location i, the memory hardware checks to see whether the page containing location i is *resident* (i.e., currently in memory). If the page is not resident, the operating system suspends the process (allowing other processes to execute), and issues a request to the disk to obtain a copy of the needed page. Once the page has been placed in main memory, the operating system makes the process ready. When the process retries the reference to location i, the reference succeeds.

10.9 Real And Virtual Address Spaces

In many operating systems, the memory manager supplies each computation with its own, independent address space. That is, an application is given a private set of memory locations that are numbered *0* through *M – 1*. The operating system works with the underlying hardware to map each address space to a set of memory locations in memory. As a result, when an application references address zero, the reference is mapped to the memory location that corresponds to zero for the process. When another application references address zero, the reference is mapped to a different location. Thus, although multiple applications can reference address zero, each reference maps to a separate location, and the applications do not interfere with one another. To be precise, we use the term *physical address space* or *real address space* to define the set of addresses that the memory hardware provides, and the term *virtual address space* to describe the set of addresses available to a given computation. At any time, memory functions in the operating system map one or more virtual address spaces onto the underlying physical address space. For example, Figure 10.1 illustrates how three virtual address spaces of *K* locations can be mapped onto an underlying physical address space that contains *3K* locations.

Figure 10.1 Illustration of three virtual address spaces mapped onto a single underlying physical address space.

From the point of view of a running computation, only addresses in the virtual address space can be referenced. Furthermore, because the system maps each address from a given virtual space to a specific region of memory, the running computation cannot accidentally read or overwrite memory that has been allocated to another computation. As a result, a system that provides each computation with its own virtual address

space helps detect programming errors and prevent problems. The point can be summarized:

> *A memory management system that maps each virtual address space into a unique block of memory prevents one computation from reading or writing memory allocated to another computation.*

In Figure 10.1, each virtual address space is smaller than the underlying physical address space. However, most memory management systems permit a virtual address space to be larger than the memory on the machine. For example, a demand paging system only keeps pages that are being referenced in main memory, and leaves copies of other pages on disk.

The original research on virtual memory arose from two motivations. First, the small size of physical memory limited the size of programs that were feasible. Second, with only one program running, a processor remained idle during I/O cycles. On the one hand, if multiple programs can be placed in memory, an operating system can switch the processor among them, allowing one to execute while another waits for an I/O operation. On the other hand, squeezing more programs into a fixed memory means each program will have less allocated memory, forcing the programs to be small. So, demand paging arose as a mechanism that handled both problems. By dividing programs into many pieces, a demand paging system can choose to keep parts of many programs in memory, keeping the processor busy when a program waits for I/O. By allowing pieces of a program's address space to be resident in memory without requiring space for the entire program, a demand paging system can support an address space that is larger than physical memory.

10.10 Hardware For Demand Paging

An operating system that maps between virtual and real addresses cannot operate without hardware support. To understand why, observe that each address, including addresses generated at runtime, must be mapped. Thus, if a program computes a value *C* and then *jumps* to location *C*, the memory system must map *C* to the corresponding real memory address. Only a hardware unit can perform the mapping efficiently.

The hardware needed for demand paging consists of a page table and an address translation unit. A page table resides in kernel memory, and there is one page table for each process. Typically, the hardware contains a register that points to the current page table and a second register that specifies the length; after it has created a page table in memory, the operating system assigns values to the registers and turns on demand paging. Similarly, when a context switch occurs, the operating system changes the page table registers to point to the page table for the new process. Figure 10.2 illustrates the arrangement.

Figure 10.2 Page tables in memory and hardware registers that specify which page table to use at a given time.

10.11 Address Translation With A Page Table

Conceptually, a page table consists of an array of pointers to memory locations. In addition to a pointer, each entry contains a bit that specifies whether the entry is valid (i.e., whether it has been initialized). *Address translation* hardware uses the current page table to translate a memory address; translation is applied to the instruction addresses as well as addresses used to fetch or store data. Translation consists of array lookup: the hardware treats the high-order bits of an address as a *page number*, uses the page number as an index into the page table, and follows the pointer to the location of the page in memory.

In practice, a page table entry does not contain a complete memory pointer. Instead, pages are restricted to start in memory locations that have zeroes in the low-order bits, and the low-order bits are omitted from each page table entry. For example, suppose a computer has 32-bit addressing and uses 4096-byte pages (i.e., each page contains 2^{12} bytes). If memory is divided into a set of 4096-byte *frames*, the starting address of each frame (i.e., the address of the first byte in the frame) will have zeroes in the 12 low-order bits. Therefore, to point to a frame in memory, a page table entry only needs to contain the upper 20 bits.

To translate an address, A, the hardware uses the upper bits of A as an index into the page table, extracts the address of a frame in memory where the page resides, and then uses the low-order bits of A as an offset into the frame. We can imagine that the translation forms a physical memory address as Figure 10.3 illustrates.

Figure 10.3 An example of virtual address translation used with paging.

Our description implies that each address translation requires a page table access (i.e., a memory access). However, such overhead would be intolerable. To make translation efficient, a processor employs a special-purpose hardware unit known as a *Translation Look-aside Buffer* (*TLB*). The TLB caches recently accessed page table entries, and can look up the answer much faster than a conventional memory access. To achieve high speed, a TLB uses a form of parallel hardware known as a *Content-Addressable Memory* (*CAM*). Given an address, CAM hardware searches a set of stored values in parallel, returning the answer in only a few clock cycles. As a result, a TLB enables a processor to run as fast with address translation turned on as it can with address translation disabled.

10.12 Metadata In A Page Table Entry

In addition to a frame pointer, each page table entry contains three bits of metadata that the hardware and operating system use. Figure 10.4 lists the bits and their meaning.

Name	Meaning
Use Bit	Set by hardware whenever the page is referenced (fetch or store)
Modify Bit	Set by hardware whenever a store operation changes data on the page
Presence Bit	Set by OS to indicate whether the page is resident in memory

Figure 10.4 The three metabits in each page table entry and their meanings.

10.13 Demand Paging And Design Questions

The term *demand paging* refers to a system where an operating system places the pages for all processes on secondary storage, and only reads a page into memory when the page is needed (i.e., on demand). Special processor hardware is required to support demand paging: if a process attempts to access a page that is not resident in memory, the hardware must suspend execution of the current instruction and notify the operating system by signaling a *page fault* exception. When a page fault occurs, the operating system finds an unused frame in memory, reads the needed page from disk, and then instructs the processor to resume the instruction that caused the fault.

When a computer first starts, memory is relatively empty, which makes finding a free frame easy. Eventually, however, all frames in memory will be filled and the operating system must select one of the filled frames, copy the page back to disk (if the page has been modified), fetch the new page, and change the page tables accordingly. The selection of a page to move back to disk forms a key problem for operating systems designers.

The design of a paging system centers on the relationship between pages and processes. When process *X* encounters a page fault, should the operating system move one of process *X*'s pages back to disk, or should the system select a page from another process? While a page is being fetched from disk, the operating system can run another process. How can the operating system ensure that at least some process has enough pages to run without also generating a page fault?† Should some pages be locked in memory? If so, which ones? How will the page selection policy interact with other policies, such as scheduling? For example, should the operating system guarantee each high-priority process a minimum number of resident pages? If the system allows processes to share memory, what policies apply to the shared pages?

One of the interesting tradeoffs in the design of a paging system arises from the balance between I/O and processing. Paging overhead and the latency a process experiences can be reduced by giving a process the maximal amount of physical memory when the process runs. However, many processes are I/O-bound, which means that a given process is likely to block waiting for I/O. When one process blocks, overall per-

†If insufficient frames exist in memory, a paging system can *thrash*, which means the frequency of page faults becomes so high that the system spends all its time paging and each process spends long periods waiting for pages.

formance is maximized if another process is ready to execute and the operating system can switch context. That is, processor utilization and overall throughput of a system can be increased by having many processes ready to execute. So the question arises: should a given process be allowed to use many frames of memory or should memory be divided among processes?

10.14 Page Replacement And Global Clock

Various page replacement policies have been proposed and tried:

- Least Recently Used (LRU)
- Least Frequently Used (LFU)
- First In First Out (FIFO)

Interestingly, a provably optimal replacement policy has been discovered. Known as *Belady's optimal page replacement algorithm*, the policy chooses to replace the page that will be referenced farthest in the future. Of course, the method is totally impractical because the operating system cannot know how pages will be used in the future. However, Belady's algorithm allows researchers to assess how well replacement policies perform.

In terms of practical systems, a single algorithm has become the de facto standard for page replacement. Known as *global clock* or *second chance*, the algorithm was devised as part of the MULTICS operating system and has relatively low overhead. The term *global* means that all processes compete with one another (i.e., when process *X* generates a page fault, the operating system can choose a frame from another process, *Y*). The alternative name of the algorithm arises because global clock is said to give used frames a "second chance" before they are reclaimed.

Global clock starts running whenever a page fault occurs. The algorithm maintains a pointer that sweeps through all the frames in memory, stopping when a free frame is found. The next time it runs, the algorithm starts at the frame just beyond where it left off.

To determine whether to select a frame, global clock checks the Use and Modify bits in the page table of the frame. If the Use/Modify bits have value (0,0), global clock chooses the frame. If the bits are (1,0), global clock resets them to (0,0) and bypasses the frame. Finally, if the bits are (1,1), global clock changes them to (1,0) and bypasses the frame, keeping a copy of the modified bit to know whether the page has been modified. In the worst case, global clock sweeps through all frames twice before reclaiming one.

In practice, most implementations use a separate process to run the global clock algorithm (which allows the clock to perform disk I/O). Furthermore, global clock does not stop immediately once a frame has been found. Instead, the algorithm continues to run, and collects a small set of candidate pages. Collecting a set allows subsequent page faults to be handled quickly and avoids the overhead associated with running the global clock algorithm frequently (i.e., avoids frequent context switching).

10.15 Perspective

Although address space management and virtual memory subsystems comprise many lines of code in an operating system, the most significant intellectual aspects of the problem arise from the choice of allocation policies and the consequent tradeoffs. Allowing each subsystem to allocate arbitrary amounts of memory maximizes flexibility and avoids the problem of a subsystem being deprived when free memory exists. Partitioning memory maximizes protection and avoids the problem of having one subsystem deprive other subsystems. Thus, a tradeoff exists between flexibility and protection.

Despite years of research, no general solution for page replacement has emerged, the tradeoffs have not been quantified, and no general guidelines exist. Similarly, despite years of research on virtual memory systems, no demand paging algorithms exist that work well for small memories. Fortunately, economics and technology have made many of the problems associated with memory management irrelevant: DRAM chip density increased rapidly, making huge memories inexpensive. As a result, computer vendors avoid memory management altogether by making the memory on each new product much larger than the memory on the previous product. Because memory is so large, the operating system can satisfy the needs of a process without taking frames from other processes. That is, a demand paging system works well not because we have devised excellent replacement algorithms, but because memories have grown so large that replacement algorithms are seldom invoked.

10.16 Summary

Low-level memory allocation mechanisms treat all of free memory as a single, exhaustible resource. High-level memory management facilities that allow memory to be partitioned into separate regions provide guarantees that prevent one subsystem from using all available memory.

The high-level memory management functions in Xinu use a buffer pool paradigm in which a fixed set of buffers is allocated in each pool. Once a pool has been created, a group of processes can allocate and free buffers dynamically. The buffer pool interface is synchronous: a given process will block until a buffer becomes available.

Large operating systems use virtual memory mechanisms to allocate a separate address space for each process. The most popular virtual memory mechanism, paging, divides the address space into fixed-size pages, and loads pages on demand. Hardware is needed to support paging because each memory reference must be mapped from a virtual address to a corresponding physical address. Paging systems only perform satisfactorily if memory is large enough to avoid running the page replacement algorithm frequently.

EXERCISES

10.1 Design a new *getmem* that subsumes *getbuf*. Hint: allow the user to suballocate from a previously allocated block of memory.

10.2 *Mkbufpool* forms a linked list of all buffers in a pool. Explain how to modify the code so it allocates memory but does not link buffers together until a call to *getbuf* requires a new buffer to be allocated.

10.3 Is *freebuf* more efficient than *freemem*? Justify your answer.

10.4 Revise the buffer pool allocation mechanism to allow deallocation of buffer pools.

10.5 The current implementation of buffer pools hides a pool ID in memory just prior to the buffer. Rewrite *freebuf* to eliminate the need for a pool ID. Ensure your version of *freebuf* will detect an invalid address (i.e., will not return a buffer to a pool unless the buffer was previously allocated from the pool).

10.6 Consider a modification of the buffer pool mechanism in which *mkbufpool* stores the pool ID in the first four bytes of a buffer and then uses the next four bytes in the buffer to store a pointer when linking the buffer onto a free list. What are the advantages of such an approach?

10.7 Suppose a processor has support for paging. Describe paging hardware that can be used to protect a process's stack from access by other processes, even if demand paging is not implemented (i.e., all pages remain resident and no replacement is performed).

10.8 Implement the scheme devised in the previous exercise to protect stacks in Xinu.

10.9 Build a demand paging version of Xinu.

10.10 Sketch the data structures an operating system needs to maintain for demand paging. Hint: think of a page replacement algorithm, such as global clock.

Chapter Contents

11.1 Introduction, 195
11.2 Inter–process Communication Ports, 195
11.3 The Implementation Of Ports, 196
11.4 Port Table Initialization, 197
11.5 Port Creation, 199
11.6 Sending A Message To A Port, 200
11.7 Receiving A Message From A Port, 202
11.8 Port Deletion And Reset, 204
11.9 Perspective, 207
11.10 Summary, 207

11

High-level Message Passing

I'm always uneasy with messages.

— Neil Tennant

11.1 Introduction

Chapter 8 describes a low-level message passing facility that permits a process to pass a message directly to another process. Although it provides a useful function, the low-level message passing system cannot be used to coordinate multiple receivers, nor can a given process participate in several message exchanges without interference among them.

This chapter completes the discussion of message passing by introducing a high-level facility that provides a synchronous interface for buffered message exchange. The message mechanism allows an arbitrary subset of processes to pass messages without affecting other processes. It introduces the concept of named rendezvous points that exist independent of processes. The implementation uses the buffer pool mechanism from the previous chapter.

11.2 Inter–process Communication Ports

Xinu uses the term *inter-process communication port* to refer to a rendezvous point through which processes can exchange messages. Message passing through ports differs from process-to-process message passing described in Chapter 8 because ports

allow multiple outstanding messages, allow multiple processes to receive from a port, and processes sending or receiving messages are blocked until requests can be satisfied. Each port is configured to hold up to a specified number of messages; each message occupies a 32-bit word. Once it has produced a message, a process can use function *ptsend* to send the message to a port. Messages are deposited in a port in FIFO order. Once a message has been sent, the sending process can continue to execute. At any time, a process can call function *ptrecv* to receive the next message from a port.

Both message sending and receiving are synchronous. As long as space remains in a port, a sender can deposit a message with no delay. Once a port becomes full, however, an additional sending process will be blocked until a message has been removed and space becomes available. Similarly, if a process tries to receive a message from an empty port, the process will be blocked until a message arrives. Requests are also handled in FIFO order. For example, if multiple processes are waiting on an empty port, the process that has been waiting the longest will receive the message. Similarly, if multiple processes are blocked waiting to send, the process that has been waiting the longest is allowed to proceed when a space becomes available.

11.3 The Implementation Of Ports

Each port consists of a queue to hold messages and two semaphores. One of the semaphores controls producers, blocking any process that attempts to add messages to a full port. The other semaphore controls consumers, blocking any process that attempts to remove a message from an empty port.

Because ports can be created dynamically, it is impossible to know the total count of items that will be enqueued at all ports at any given time. Although each message is small (one word), the total space required for port queues must be limited to prevent the port functions from using all the free space. To guarantee a limit on the total space used, the port functions allocate a fixed number of nodes to hold messages, and share the set among all ports. Initially, the message nodes are linked into a free list given by variable *ptfree*. Function *ptsend* removes a node from the free list, stores the message in the node, and adds the node to the queue associated with the port to which the message has been sent. Function *ptrecv* extracts the next message from a specified port, returns the node containing the message to the free list, and delivers the message to its caller.

Structure *ptnode* in file *ports.h* defines the contents of a node that contains one message. A *ptnode* structure contains two fields: *ptmsg* holds a 32-bit message, and *ptnext* points to the next message.

Structure *ptentry* defines the contents of an entry in the port table. Fields *ptssem* and *ptrsem* contain the IDs of semaphores that control sending and receiving. Field *ptstate* specifies whether the entry is currently being used, and field *ptmaxcnt* specifies the maximum messages that are allowed in the port at any time. Fields *pthead* and *pttail* point to the first node on the message list and the last, respectively. We will discuss the sequence field, *ptseq*, later.

```
/* ports.h - isbadport */

#define NPORTS          30              /* Maximum number of ports      */
#define PT_MSGS         100             /* Total messages in system     */
#define PT_FREE         1               /* Port is free                 */
#define PT_LIMBO        2               /* Port is being deleted/reset  */
#define PT_ALLOC        3               /* Port is allocated            */

struct  ptnode  {                       /* Node on list of messages     */
        uint32  ptmsg;                  /* A one-word message           */
        struct  ptnode  *ptnext;        /* Pointer to next node on list */
};

struct  ptentry {                       /* Entry in the port table      */
        sid32   ptssem;                 /* Sender semaphore             */
        sid32   ptrsem;                 /* Receiver semaphore           */
        uint16  ptstate;                /* Port state (FREE/LIMBO/ALLOC)*/
        uint16  ptmaxcnt;               /* Max messages to be queued    */
        int32   ptseq;                  /* Sequence changed at creation */
        struct  ptnode  *pthead;        /* List of message pointers     */
        struct  ptnode  *pttail;        /* Tail of message list         */
};

extern  struct  ptnode  *ptfree;        /* List of free nodes           */
extern  struct  ptentry porttab[];      /* Port table                   */
extern  int32   ptnextid;               /* Next port ID to try when     */
                                        /*   looking for a free slot    */

#define isbadport(portid)       ( (portid)<0 || (portid)>=NPORTS )
```

11.4 Port Table Initialization

Because initialization code is designed after basic operations have been implemented, we have been discussing initialization functions after other functions. In the case of ports, however, we will discuss initialization first, because doing so will make the remaining functions easier to understand. File *ptinit.c* contains the code to initialize ports and a declaration of the port table. Global variable *ptnextid* gives the index in array *porttab* at which the search will start when a new port is needed. Initialization consists of marking each port free, forming the linked list of free nodes, and initializing *ptnextid*. To create a free list, *ptinit* uses *getmem* to allocate a block of memory, and then moves through the memory, linking individual message nodes together to form a free list.

```
/* ptinit.c - ptinit */

#include <xinu.h>

struct  ptnode  *ptfree;                /* List of free message nodes   */
struct  ptentry porttab[NPORTS];        /* Port table                   */
int32   ptnextid;                       /* Next table entry to try      */

/*------------------------------------------------------------------------
 *  ptinit  -  Initialize all ports
 *------------------------------------------------------------------------
 */
syscall ptinit(
          int32 maxmsgs                 /* Total messages in all ports  */
        )
{
        int32   i;                      /* Runs through the port table  */
        struct  ptnode  *next, *curr;   /* Used to build a free list    */

        /* Allocate memory for all messages on all ports */

        ptfree = (struct ptnode *)getmem(maxmsgs*sizeof(struct ptnode));
        if (ptfree == (struct ptnode *)SYSERR) {
                panic("pinit - insufficient memory");
        }

        /* Initialize all port table entries to free */

        for (i=0 ; i<NPORTS ; i++) {
                porttab[i].ptstate = PT_FREE;
                porttab[i].ptseq = 0;
        }
        ptnextid = 0;

        /* Create a free list of message nodes linked together */

        for ( curr=next=ptfree ;  --maxmsgs > 0  ; curr=next ) {
                curr->ptnext = ++next;
        }

        /* Set the pointer in the final node to NULL */

        curr->ptnext = NULL;
        return OK;
}
```

11.5 Port Creation

Port creation consists of allocating an entry in the port table from among those that are free. Function *ptcreate* performs the allocation and returns a *port identifier* (*port ID*) to its caller. *Ptcreate* takes an argument that specifies the maximum count of outstanding messages that the port will allow. Thus, when a port is created, the calling process can determine how many messages can be enqueued on the port before any sender blocks.

```
/* ptcreate.c - ptcreate */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ptcreate  -  Create a port that allows "count" outstanding messages
 *------------------------------------------------------------------------
 */
syscall ptcreate(
          int32         count           /* Size of port                 */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        int32   i;                      /* Counts all possible ports    */
        int32   ptnum;                  /* Candidate port number to try */
        struct  ptentry *ptptr;         /* Pointer to port table entry  */

        mask = disable();
        if (count < 0) {
                restore(mask);
                return SYSERR;
        }

        for (i=0 ; i<NPORTS ; i++) {    /* Count all table entries      */
                ptnum = ptnextid;       /* Get an entry to check        */
                if (++ptnextid >= NPORTS) {
                        ptnextid = 0;   /* Reset for next iteration     */
                }

                /* Check table entry that corresponds to ID ptnum */

                ptptr= &porttab[ptnum];
                if (ptptr->ptstate == PT_FREE) {
                        ptptr->ptstate = PT_ALLOC;
                        ptptr->ptssem = semcreate(count);
```

```
                        ptptr->ptrsem = semcreate(0);
                        ptptr->pthead = ptptr->pttail = NULL;
                        ptptr->ptseq++;
                        ptptr->ptmaxcnt = count;
                        restore(mask);
                        return ptnum;
                }
        }
        restore(mask);
        return SYSERR;
}
```

11.6 Sending A Message To A Port

The basic operations on ports, sending and receiving messages, are handled by functions *ptsend* and *ptrecv*. Each requires the caller to specify the port on which the operation is to be performed by passing a port ID as an argument. Function *ptsend* adds a message to those that are waiting at the port. It waits for space in the port, enqueues the message given by its argument, signals the receiver semaphore to indicate another message is available, and returns. The code is found in file *ptsend.c*:

```
/* ptsend.c - ptsend */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ptsend  -  Send a message to a port by adding it to the queue
 *------------------------------------------------------------------------
 */
syscall ptsend(
          int32         portid,         /* ID of port to use            */
          umsg32        msg             /* Message to send              */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  ptentry *ptptr;         /* Pointer to table entry       */
        int32   seq;                    /* Local copy of sequence num.  */
        struct  ptnode  *msgnode;       /* Allocated message node       */
        struct  ptnode  *tailnode;      /* Last node in port or NULL    */

        mask = disable();
        if ( isbadport(portid) ||
             (ptptr= &porttab[portid])->ptstate != PT_ALLOC ) {
```

```
                restore(mask);
                return SYSERR;
        }

        /* Wait for space and verify port has not been reset */

        seq = ptptr->ptseq;             /* Record orignal sequence      */
        if (wait(ptptr->ptssem) == SYSERR
            || ptptr->ptstate != PT_ALLOC
            || ptptr->ptseq != seq) {
                restore(mask);
                return SYSERR;
        }
        if (ptfree == NULL) {
                panic("Port system ran out of message nodes");
        }

        /* Obtain node from free list by unlinking */

        msgnode = ptfree;               /* Point to first free node     */
        ptfree  = msgnode->ptnext;      /* Unlink from the free list    */
        msgnode->ptnext = NULL;         /* Set fields in the node       */
        msgnode->ptmsg  = msg;

        /* Link into queue for the specified port */

        tailnode = ptptr->pttail;
        if (tailnode == NULL) {         /* Queue for port was empty     */
                ptptr->pttail = ptptr->pthead = msgnode;
        } else {                        /* Insert new node at tail      */
                tailnode->ptnext = msgnode;
                ptptr->pttail = msgnode;
        }
        signal(ptptr->ptrsem);
        restore(mask);
        return OK;
}
```

The initial code in *ptsend* merely verifies that argument *portid* specifies a valid port ID. What happens next is more interesting. *Ptsend* makes a local copy of the sequence number, *ptseq*, and then processes the request. It waits on the sender semaphore, and then verifies that the port is still allocated and that the sequence number agrees. It may seem odd that *ptsend* verifies the port ID a second time. However, if the port is already full when *ptsend* runs, the calling process can be blocked. Further-

more, while the process is blocked waiting to send, the port may be deleted (and even recreated). To understand how the sequence number helps, recall that *ptcreate* increments the sequence number when a port is created. The idea is to have waiting processes verify that the *wait* did not terminate because the port was deleted. If it did, the port will either remain unused or the sequence number will be incremented. Thus, the code following the call to *wait* verifies that the original port remains allocated.

The implementation of *ptsend* enqueues messages in FIFO order. It relies on *pttail* to point to the last node on the queue if the queue is nonempty. Furthermore, *ptsend* always makes *pttail* point to a new node after the node has been added to the list. Finally, *ptsend* signals the receiver semaphore after the new message has been added to the queue, allowing a receiver to consume the message.

As in earlier code, an invariant helps a programmer understand the implementation. The invariant states:

> *Semaphore* ptrsem *has a nonnegative count* n *if* n *messages are waiting in the port; it has negative count* –n *if* n *processes are waiting for messages*.

The call to *panic* also deserves comment because this is its first occurrence. In our design, running out of message nodes is a catastrophe from which the system cannot recover. It means that the arbitrary limit on message nodes, set to prevent ports from using all the free memory, is insufficient. Perhaps the processes using ports are performing incorrectly. Perhaps the request is valid, but the message limit was set too low; there is no way to know. Under such circumstances it is often better to announce failure and stop rather than attempt to go on. In our system, *panic* handles such situations by printing the specified error message and halting the processor. Panic could be changed to reboot the entire system. (The exercises suggest alternative ways of handling the problem.)

11.7 Receiving A Message From A Port

Function *ptrecv* implements a basic consumer operation. It removes a message from a specified port and returns the message to its caller. The code is found in file *ptrecv.c*:

```
/* ptrecv.c - ptrecv */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ptrecv  -  Receive a message from a port, blocking if port empty
 *------------------------------------------------------------------------
```

```
 */
uint32  ptrecv(
          int32         portid          /* ID of port to use            */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  ptentry *ptptr;         /* Pointer to table entry       */
        int32   seq;                    /* Local copy of sequence num.  */
        umsg32  msg;                    /* Message to return            */
        struct  ptnode  *msgnode;       /* First node on message list   */

        mask = disable();
        if ( isbadport(portid) ||
             (ptptr= &porttab[portid])->ptstate != PT_ALLOC ) {
                restore(mask);
                return (uint32)SYSERR;
        }

        /* Wait for message and verify that the port is still allocated */

        seq = ptptr->ptseq;             /* Record orignal sequence      */
        if (wait(ptptr->ptrsem) == SYSERR || ptptr->ptstate != PT_ALLOC
            || ptptr->ptseq != seq) {
                restore(mask);
                return (uint32)SYSERR;
        }

        /* Dequeue first message that is waiting in the port */

        msgnode = ptptr->pthead;
        msg = msgnode->ptmsg;
        if (ptptr->pthead == ptptr->pttail)     /* Delete last item     */
                ptptr->pthead = ptptr->pttail = NULL;
        else
                ptptr->pthead = msgnode->ptnext;
        msgnode->ptnext = ptfree;               /* Return to free list  */
        ptfree = msgnode;
        signal(ptptr->ptssem);
        restore(mask);
        return msg;
}
```

Ptrecv checks its argument, waits until a message is available, verifies that the port has not been deleted or reused, and dequeues the message node. It records the value of

the message in local variable *msg* before returning the message node to the free list, and then returns the value to its caller.

11.8 Port Deletion And Reset

It is sometimes useful to delete or to reset a port. In either case, the system must dispose of waiting messages (if the port contains messages), return message nodes to the free list, and permit waiting processes to continue execution. How should the port system dispose of waiting messages? It could choose to throw them away, or to return them to the processes that sent them. To permit maximum flexibility, our design does not make a choice. Instead, it permits a caller to specify how to dispose of messages. Functions *ptdelete* and *ptreset* perform port deletion and reset operations. Each function takes an argument that specifies a function to be called to dispose of each waiting message. The code is found in files *ptdelete.c* and *ptreset.c*:

```
/* ptdelete.c - ptdelete */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ptdelete  -  Delete a port, freeing waiting processes and messages
 *------------------------------------------------------------------------
 */
syscall ptdelete(
          int32         portid,         /* ID of port to delete         */
          int32         (*disp)(int32)  /* Function to call to dispose  */
        )                               /*    of waiting messages       */
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  ptentry *ptptr;         /* Pointer to port table entry  */

        mask = disable();
        if ( isbadport(portid) ||
             (ptptr= &porttab[portid])->ptstate != PT_ALLOC ) {
                restore(mask);
                return SYSERR;
        }
        _ptclear(ptptr, PT_FREE, disp);
        ptnextid = portid;
        restore(mask);
        return OK;
}
```

```
/* ptreset.c - ptreset */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ptreset  -  Reset a port, freeing waiting processes and messages and
                        leaving the port ready for further use
 *------------------------------------------------------------------------
 */
syscall ptreset(
          int32         portid,         /* ID of port to reset          */
          int32         (*disp)(int32)  /* Function to call to dispose  */
        )                               /*   of waiting messages        */
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  ptentry *ptptr;         /* Pointer to port table entry  */

        mask = disable();
        if ( isbadport(portid) ||
             (ptptr= &porttab[portid])->ptstate != PT_ALLOC ) {
                restore(mask);
                return SYSERR;
        }
        _ptclear(ptptr, PT_ALLOC, disp);
        restore(mask);
        return OK;
}
```

Both *ptdelete* and *ptreset* verify that their arguments are correct, and then call *_ptclear* to perform the work of clearing messages and waiting processes.† The disposal function can cause a context switch, which means other processes can run and use ports. Therefore, while it proceeds to clear a port, *_ptclear* places the port in a "limbo" state (*PT_LIMBO*). The limbo state guarantees that no other processes can use the port — functions *ptsend* and *ptrecv* will refuse to operate on a port that is not allocated, and *pcreate* will not allocate the port unless it is free. Thus, even if the disposal function switches context, no problems will occur.

Before declaring a port eligible for use again, *_ptclear* calls *dispose* repeatedly, passing it each waiting message. Finally, after all messages have been removed, *_ptclear* deletes or resets the semaphores as specified by its second argument. Before disposing of messages, *_ptclear* increments the port sequence number so that waiting processes can tell that the port has changed when they awaken. The code is found in file *ptclear.c*:

†The name *_ptclear* begins with an underscore to indicate that the function is internal to the system and is not intended to be called by an application process (i.e., is not part of the system's API).

```
/* ptclear.c - _ptclear */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  _ptclear  -  Used by ptdelete and ptreset to clear or reset a port
 *                 (internal function assumes interrupts disabled and
 *                 arguments have been checked for validity)
 *------------------------------------------------------------------------
 */
void    _ptclear(
          struct ptentry *ptptr,        /* Table entry to clear         */
          uint16       newstate,        /* New state for port           */
          int32        (*dispose)(int32)/* Disposal function to call    */
         )
{
        struct  ptnode  *walk;          /* Pointer to walk message list */

        /* Place port in limbo state while waiting processes are freed */

        ptptr->ptstate = PT_LIMBO;

        ptptr->ptseq++;                 /* Reset accession number       */
        walk = ptptr->pthead;           /* First item on msg list       */

        if ( walk != NULL ) {           /* If message list nonempty     */

                /* Walk message list and dispose of each message */

                for( ; walk!=NULL ; walk=walk->ptnext) {
                        (*dispose)( walk->ptmsg );
                }

                /* Link entire message list into the free list */

                (ptptr->pttail)->ptnext = ptfree;
                ptfree = ptptr->pthead;
        }

        if (newstate == PT_ALLOC) {
                ptptr->pttail = ptptr->pthead = NULL;
                semreset(ptptr->ptssem, ptptr->ptmaxcnt);
                semreset(ptptr->ptrsem, 0);
        } else {
                semdelete(ptptr->ptssem);
```

```
                semdelete(ptptr->ptrsem);
        }
        ptptr->ptstate = newstate;
        return;
}
```

11.9 Perspective

Because it provides a synchronous interface, our port mechanism allows a process to wait for the next message to arrive. A synchronous interface can be powerful — a clever programmer can use the mechanism to coordinate processes (e.g., to implement mutual exclusion). Interestingly, the ability to coordinate processes also introduces a potential problem: deadlock. That is, a set of processes that uses ports to exchange messages can end up with all processes in the set blocked, waiting for a message to arrive with no processes left running to send a message. Therefore, a programmer using ports must be careful to guarantee that such deadlocks cannot occur.

11.10 Summary

The chapter introduces a high-level message passing mechanism, called communication ports, that permits processes to exchange messages through rendezvous points. Each port consists of a fixed length queue of messages. Function *ptsend* deposits a message at the tail of the queue, and function *ptrecv* extracts an item from the head of the queue. Processes that attempt to receive from an empty port are blocked until a message arrives; processes that attempt to send to a full port are blocked until space becomes available.

EXERCISES

11.1 Consider the primitives *send—receive* and *ptsend—ptrecv*. Is it possible to design a single message passing scheme that encompasses both? Explain.

11.2 An important distinction is made between statically allocated and dynamically allocated resources. For example, ports are dynamically allocated while inter-process message slots are statically allocated. What is the key problem with dynamic allocation in a multi-process environment?

11.3 Change the message node allocation scheme so that a semaphore controls nodes on the free list. Have *ptsend* wait for a free node if none exists. What potential problems, if any, does the new scheme introduce?

11.4 *Panic* is used for conditions like internal inconsistency or potential deadlock. Often the conditions causing a panic are irreproducible, so their cause is difficult to pinpoint. Discuss what you might do to trace the cause of the panic in *ptsend*.

11.5 As alternatives to the call to *panic* in *ptsend*, consider allocating more nodes or retrying the operation. What are the liabilities of each?

11.6 Rewrite *ptsend* and *ptrecv* to return a special value when the port is deleted while they are waiting. What is the chief disadvantage of the new mechanism?

11.7 Modify the routines in previous chapters that allocate, use, and delete objects so they use sequence numbers to detect deletion as the communication port functions do.

11.8 *Ptsend* and *ptrecv* cannot transmit a message with value equal to *SYSERR* because *ptrecv* cannot distinguish between a message with that value and an error. Redesign the functions to transmit any value.

11.9 Although each call to *ptcreate* specifies a count of messages, the implementation does not check that the set of allocated message nodes is sufficient to handle all possible requests. Modify *ptcreate* to ensure that the sum of messages on all ports is less than *PT_MSGS*.

11.10 If the previous exercise is implemented, what important change can be made to *ptsend*?

Chapter Contents

12.1 Introduction, 211
12.2 The Advantage Of Interrupts, 212
12.3 Interrupt Processing, 212
12.4 Vectored Interrupts, 213
12.5 Integration Of Interrupts And Exceptions, 214
12.6 ARM Exception Vectors Using Code, 215
12.7 Assignment Of Device Interrupt Vector Numbers, 219
12.8 Interrupt Dispatching, 220
12.9 The Structure Of Interrupt Software, 221
12.10 Disabling Interrupts, 223
12.11 Constraints On Functions That Interrupt Code Invokes, 225
12.12 The Need To Reschedule During An Interrupt, 225
12.13 Rescheduling During An Interrupt, 226
12.14 Perspective, 227
12.15 Summary, 228

12

Interrupt Processing

The joy of music should never be interrupted by a commercial.

— Leonard Bernstein

12.1 Introduction

Previous chapters focus on processor and memory management. The chapters on processor management introduce the concept of concurrent processing, show how processes are created and terminated, and describe how processes coordinate. The chapters on memory management illustrate low-level mechanisms used to manage dynamic allocation and the release of stack and heap storage.

This chapter begins a discussion of input and output (I/O) facilities. The chapter reviews the concept of an interrupt, and introduces the overall software architecture that an operating system uses to handle interrupts. It describes how an operating system passes control to the appropriate interrupt handler software when an interrupt occurs. More important, the chapter explains the complex relationship between interrupts and the operating system abstraction of concurrent processes, and gives general guidelines that interrupt code must follow to provide a correct and safe implementation of concurrent processes in the presence of interrupts. Later chapters continue the discussion by examining specific devices, including a real-time clock that enables preemptive process scheduling. The chapters explain how interrupt handler software forms part of a driver.

12.2 The Advantage Of Interrupts

The interrupt mechanism invented for third-generation computer systems provides a powerful facility that separates I/O activities from processing. Many of the services an operating system offers would not be possible without hardware that supports interrupts.

The motivation for an interrupt mechanism arises from the observation that I/O hardware can function in parallel with processing. Instead of relying on the processor to provide complete control over I/O, each I/O device contains hardware that can operate independently. The processor only needs to start or stop a device — once started, a device proceeds to transfer data without further help. Because most I/O proceeds much slower than computation, the processor can start multiple devices and allow the operations to proceed in parallel. After starting I/O, the processor can perform other computation (i.e., execute a process) until a device interrupts to signal completion. The key idea is:

> *An interrupt mechanism permits the processor and I/O devices to operate in parallel. Although the details differ, the hardware includes a mechanism that automatically "interrupts" normal processing and informs the operating system when a device completes an operation or needs attention.*

12.3 Interrupt Processing

Hardware in a processor performs three basic steps when an interrupt occurs:

- Set the hardware to prevent further interrupts from occurring while an interrupt is being processed
- Save sufficient state to allow the processor to resume execution transparently once the interrupt has been handled
- Branch to a predetermined memory location where the operating system has placed code to process the interrupt

Each processor includes details that complicate interrupt processing. For example, when it saves state, the hardware in most systems does not save a complete copy of all processor registers. Instead, the hardware records a few basic values, such as a copy of the *instruction pointer*,† and requires the operating system to save any other registers that will be used during interrupt processing. To ensure that interrupt processing does not change any values visible to the process that was executing, the operating system must restore all the saved registers before returning to normal processing once the interrupt handling is complete.

---

†The instruction pointer, sometimes called a *program counter*, contains the address of the instruction to be executed next.

12.4 Vectored Interrupts

When an interrupt occurs, the operating system must have a way to identify which device caused the interrupt. A variety of hardware mechanisms have been invented that handle device identification. For example, on some systems, the processor uses the bus to ask the interrupting device to identify itself. On other systems, a separate controller unit handles communication with devices. Some designs use a separate co-processor instead of a controller. After considering other aspects of interrupt handling, we will see how the example platforms handle interrupts.

How should a device identify itself? The most popular technique is known as a *vectored interrupt*. Each device is assigned a unique integer, 0, 1, 2, and so on. Using terminology that has become popular, the integer is called an *Interrupt Request Number* (*IRQ*). When an interrupt occurs, the device specifies its IRQ. A vectored interrupt mechanism helps make interrupt processing extremely efficient: the IRQ can be used as an index into an *interrupt vector array*. The operating system preloads each location in the interrupt vector with a pointer to a function that handles the interrupt. When a device with IRQ *i* interrupts, control branches to:

interrupt_vector[i]

Figure 12.1 illustrates the conceptual organization of interrupt vectors as an array of pointers.

Figure 12.1 The conceptual organization of interrupt vectors as an array of pointers, where each entry gives the address of code that handles the interrupt for a specific device.

12.5 Integration Of Interrupts And Exceptions

Although the vectored approach was invented to handle interrupts from I/O devices, the scheme has been extended to handle *exceptions*. That is, the vector can have additional slots that are devoted to handle error conditions, such as page faults, divide-by-zero errors, protection violations, illegal instructions, and illegal memory references. The processor hardware is constructed such that it generates an exception when an error occurs. In fact, exception processing is so fundamental that some platforms use the term *exception vector* to refer to the array of pointers instead of the term *interrupt vector*. The two example platforms illustrate two approaches vendors have used to combine exceptions and interrupts.

12.5.1 Galileo (Intel)

The Galileo uses a single, large vector that contains entries for both exceptions and device interrupts. The first entries in the array correspond to exceptions, and devices are assigned IRQ values above the exceptions. For example, vector position zero is reserved for arithmetic division errors. When a program attempts to divide by zero, the hardware detects the problem and raises *exception 0*. In essence, the hardware behaves exactly as if an interrupt had occurred with IRQ 0 (i.e., the program counter is saved and control passes to the address given by interrupt_vector[0]). Similarly, if it encounters an illegal opcode, the processor raises exception 5 exactly as if an interrupt had occurred with IRQ 5.

12.5.2 BeagleBone Black (ARM)

The BeagleBone Black uses a two-level approach. At the top level, the processor defines eight possible exceptions: *Reset*, *Undefined instruction*, *Software interrupt*, *Pre-Fetch Abort*, *Data Abort*, *Unused*, *IRQ*, and *FIQ*. All errors, such as divide-by-zero, are classified as one of the exceptions. Two of the exceptions (*IRQ* and *FIQ*) correspond to I/O device interrupts.† When a device generates an IRQ interrupt, the hardware raises an IRQ exception. The processor uses an exception vector to branch to the code that handles the exception. That is, all device interrupts follow the same exception vector pointer. The code that handles the IRQ exception must interrogate the hardware, obtain the IRQ for the interrupting device, and then use a second level of indirection to find the code that handles the specific device. The point is:

> *Exceptions, such as divide-by-zero and page faults, follow a vectored approach. The Galileo illustrates how interrupts and exceptions are integrated into a single* exception vector. *The BeagleBone Black illustrates a two-level scheme in which device interrupts correspond to one particular exception and the operating system must use a second level of indirection to reach the handler for a specific device.*

†IRQ corresponds to a normal interrupt; FIQ can be used to optimize interrupt processing for device hardware that does not need the processor to save as much state.

12.6 ARM Exception Vectors Using Code

The ARM architecture adds one more wrinkle to exception vector processing: instead of containing pointers to handlers as the Intel architecture does, an ARM exception vector contains instructions that the processor executes. When exception *i* occurs, the processor starts executing the code at:

exception_vector[i]

Each location in the exception vector is only one instruction long (i.e., four bytes), which means the instruction must branch to the code for the exception. The instruction in the exception vector for device interrupts jumps to the device dispatch code, the instruction in the exception vector for software interrupts jumps to the software interrupt code, and so on.

An operating system must initialize exception vectors, and may need to change the vectors later. Because ARM exception vectors contain code rather than data, changing the contents means creating an executable instruction, which can be dangerous. To avoid modifying code at runtime, Xinu uses *indirect branch* instructions. That is, Xinu defines an additional array of pointers that is parallel to the exception vectors, and makes each exception vector contain an indirect jump instruction that references the corresponding location in the parallel array of pointers. Figure 12.2 illustrates the concept.

Figure 12.2 An ARM exception vector with instructions that each jump indirectly through an item in an array of pointers.

Using an extra array has both a practical and conceptual advantage. It is practical because the operating system only needs to change a pointer to change the function associated with a given exception; there is no need to create executable instructions at runtime. The conceptual advantage should also be clear: the scheme makes the ARM exception mechanism match the model presented in Figure 12.1 on page 213.

How is the parallel array mechanism from Figure 12.2 implemented? Because a normal *jump* instruction takes two words, Xinu places the parallel array immediately after the exception vectors, and uses *PC-relative* addressing, which makes the *jump* fit into a single word. More important, because the ith location of the parallel array is a constant distance from the ith exception vector, every exception vector location contains the same offset in the *jump* instruction.† Figure 12.3 illustrates the implementation.

Figure 12.3 ARM exception vectors and an indirect array are contiguous in memory.

Function *initevec* near the end of file intr.S assigns the relative jump instruction to locations in the exception vector, assigns the address of a default exception handler to each position in the parallel array, and then changes the pointer associated with IRQs to point to the IRQ exception hander, *irq_except*.

```
/* intr.S - enable, disable, restore, halt, pause, irq_except (ARM) */

#include <armv7a.h>

        .text
        .globl  disable
        .globl  restore
        .globl  enable
        .globl  pause
        .globl  halt
        .globl  irq_except
        .globl  irq_dispatch
        .globl  initevec
        .globl  expjmpinstr

/*------------------------------------------------------------------------
 * disable  -  Disable interrupts and return the previous state
 *------------------------------------------------------------------------
 */
disable:
        mrs     r0, cpsr        /* Copy the CPSR into r0                */
        cpsid   i               /* Disable interrupts                   */
        mov     pc, lr          /* Return the CPSR                      */
```

†The offset in the indirect jump is 24 bytes instead of 32 bytes because the program counter points two words beyond the instruction being executed.

```
/*-----------------------------------------------------------------------
 * restore  -  Restore interrupts to value given by mask argument
 *-----------------------------------------------------------------------
 */
restore:
        push    {r1, r2}        /* Save r1, r2 on stack                 */
        mrs     r1, cpsr        /* Copy CPSR into r1                    */
        ldr     r2, =0x01F00220
        and     r1, r1, r2      /* Extract flags and other important    */
        bic     r0, r0, r2      /*    bits from the mask                */
        orr     r1, r1, r0
        msr     cpsr_cfsx, r1   /* Restore the CPSR                     */
        pop     {r1, r2}        /* Restore r1, r2                       */
        mov     pc, lr          /* Return to caller                     */

/*-----------------------------------------------------------------------
 * enable  -  Enable interrupts
 *-----------------------------------------------------------------------
 */
enable:
        cpsie   i               /* Enable interrupts                    */
        mov     pc, lr          /* Return                               */

/*-----------------------------------------------------------------------
 * pause or halt  -  Place the processor in a hard loop
 *-----------------------------------------------------------------------
 */
halt:
pause:
        cpsid   i               /* Disable interrupts           */
dloop:  b       dloop           /* Dead loop                    */

/*-----------------------------------------------------------------------
 * irq_except  -  Dispatch an IRQ exception to higher level IRQ dispatcher
 *-----------------------------------------------------------------------
 */
irq_except:
        sub     lr, lr, #4      /* Correct the return address           */
        srsdb   sp!, #19        /* Save return state on the supervisor  */
                                /*   mode stack                         */
        cps     #19             /* Change to supervisor mode            */
        push    {r0-r12, lr}    /* Save all registers                   */
        bl      irq_dispatch    /* Call IRQ dispatch                    */
        pop     {r0-r12, lr}    /* Restore all registers                */
        rfeia   sp!             /* Return from the exception using info */
                                /*   stored on the stack                */
```

```
/*-----------------------------------------------------------------------
 * defexp_handler - Default Exception handler
 *-----------------------------------------------------------------------
 */
defexp_handler:
        ldr     r0, =expmsg1
        mov     r1, lr
        bl      kprintf
        ldr     r0, =expmsg2
        bl      panic

/*-----------------------------------------------------------------------
 * initevec - Initialize the exception vector
 *-----------------------------------------------------------------------
 */
initevec:
        mrc     p15, 0, r0, c1, c0, 0 /* Read the c1-control register   */
        bic     r0, r0, #ARMV7A_C1CTL_V/* V bit = 0, normal exp. base   */
        mcr     p15, 0, r0, c1, c0, 0 /* Write the c1-control register  */
        ldr     r0, =ARMV7A_EV_START  /* Exception base address         */
        mcr     p15, 0, r0, c12, c0, 0/* Store excp. base addr. in c12  */
        ldr     r0, =ARMV7A_EV_START  /* Start address of exp. vector   */
        ldr     r1, =ARMV7A_EV_END    /* End address of exp. vector     */
        ldr     r2, =expjmpinstr      /* Copy the exp jump instr        */
        ldr     r2, [r2]              /*   into register r2             */
expvec: str     r2, [r0]              /* Store the jump instruction     */
        add     r0, r0, #4            /*   in the exception vector      */
        cmp     r0, r1
        bne     expvec
        ldr     r0, =ARMV7A_EH_START  /* Install the default exception  */
        ldr     r1, =ARMV7A_EH_END    /*   handler for all exceptions   */
        ldr     r2, =defexp_handler
exphnd: str     r2, [r0]
        add     r0, r0, #4
        cmp     r0, r1
        bne     exphnd
        ldr     r0, =ARMV7A_IRQH_ADDR /* Install the IRQ handler to     */
        ldr     r1, =irq_except       /*   override the default         */
        str     r1, [r0]              /*   exception handler            */
        mov     pc, lr

/*-----------------------------------------------------------------------
 * expjmpinstr - A PC relative jump instruction, copied into exp. vector
 *-----------------------------------------------------------------------
 */
```

```
expjmpinstr:
        ldr     pc, [pc, #24]
```

The last statement of the file is the relative jump instruction. We use the assembler to create the instruction, which is both easier to read and less prone to errors than defining a constant.

12.7 Assignment Of Device Interrupt Vector Numbers

The position of exceptions in an exception vector are chosen when a system is designed and never change. For example, we said that the Galileo hardware is built so an illegal instruction error always raises exception 5. On the BeagleBone Black, an illegal instruction raises exception 6. However, IRQ values cannot be preassigned unless the set of devices is fixed when the hardware is built (e.g., an SoC that has three devices). To understand why, observe that most computer systems allow an owner to purchase and install new device hardware. To accommodate an arbitrary set of devices, three basic approaches have been used for IRQ assignment:

- Manual device configuration
- Automated assignment during bootstrap
- Dynamic assignment for pluggable devices

Manual device configuration. On early hardware, a human had to assign a unique IRQ to each device before the device was connected to a computer. Typically, the assignment was made using switches or wire jumpers on the device circuit board. Once an assignment was made to the hardware, the operating system had to be configured to match the hardware. Manual assignment had the problems of being tedious and error-prone — if a human accidentally assigned the same IRQ to two different devices, or the vector number configured into the operating system did not match the IRQ value configured into device hardware, devices would fail to operate correctly.

Automated assignment during bootstrap. As bus hardware became more sophisticated, techniques were developed that automated interrupt vector assignments. Automated assignment requires *programmable* devices. That is, the operating system uses the bus to find devices that are attached to the bus, and assigns each device an IRQ. In essence, programmable devices allow the paradigm to be reversed: instead of assigning an IRQ to a device and then configuring the operating system to match the assignment, programmable devices allow the operating system to choose an IRQ and then assign the number to the device. Because the operating system handles assignment when a computer boots, the automated approach eliminates human error, and makes it possible for users to attach new hardware to their computer without understanding IRQ assignment.

Dynamic assignment for pluggable devices. A final approach is used to accommodate devices that can be plugged into a computer while the operating system is running.

For example, consider a *Universal Serial Bus* (*USB*) that allows devices to be connected or disconnected at runtime. A USB uses two levels of interrupt binding. At the first level, a computer contains a USB host controller device that attaches to the computer's bus. When it boots, the operating system recognizes the host controller, and assigns it a unique IRQ value using the same automated assignment used for other devices. The operating system configures a device driver for the USB host controller, and the driver includes a function that handles interrupts. We will use the term *master driver* to describe the software that handles USB host controller interrupts.

At runtime, when a user plugs a device into a USB port, a second level binding occurs. The USB controller hardware detects the new device, and generates an interrupt. The master driver receives control, and communicates across the USB to obtain the type and model of the new device. The master dynamically loads a driver for the new device, and records the driver location. Later, when the USB device interrupts, the master obtains control, determines which of the devices plugged into the USB generated the interrupt, and forwards control to the appropriate driver. Finally, when a user disconnects a USB device, the master receives an interrupt, determines which device has been unplugged, and removes its record of the device.

12.8 Interrupt Dispatching

Several questions arise about the design of an interrupt and/or exception mechanism. How large is an interrupt or exception vector? Does an entry in a vector contain more than the address of a handler? Where is the vector stored? Does the hardware or operating system perform indexing? A variety of designs have been used. Most systems choose a maximum size for the vector. Some systems place the vector at a fixed location in memory; a slightly more advanced design allows the operating system to choose a memory location and then inform the hardware.

We use the term *interrupt dispatching* to refer to the steps that are taken to obtain a device number, use the device number as an index into a vector, extract the address of a handler, and pass control to the handler. To make dispatching efficient and to separate I/O details from the processor, many systems use a special-purpose hardware device known as an *interrupt controller*.†

The example platforms illustrate two types of interrupt controllers. The controller used on the BeagleBone Black handles communication over the bus, but does not store interrupt vectors. Instead, when an interrupt occurs, the controller obtains an integer IRQ from the device, raises an IRQ exception, and passes the IRQ value to the processor. The operating system manages the interrupt vectors, and performs the steps of using the IRQ as an index into the interrupt vector and passing control to the appropriate handler. The interrupt controller for the Galileo stores the entire interrupt vector. When a device interrupts, the controller obtains the IRQ, uses the IRQ as an index into the interrupt vector array, and calls the appropriate handler. On the Galileo, the operating system must load the vector into the controller at startup, before any interrupts or exceptions occur. The point is:

---

†In a traditional computer design, an interrupt controller consists of a separate chip; in an SoC design, the controller is physically located on the SoC, but remains external to the processor.

Most systems use external interrupt controller hardware. The design of the controller determines which steps of interrupt dispatching are handled by the hardware and which steps are handled by the operating system.

12.9 The Structure Of Interrupt Software

Because interrupt dispatching involves saving and restoring hardware state, dispatch code must be written in assembly language. However, system designers prefer to write device drivers, including interrupt handlers, in a high-level language, such as C. To accommodate the requirement for assembly language, we follow an approach used in most systems: interrupt code is divided into two parts: a low-level piece written in assembly language and a high-level piece written in C. We use the term *dispatcher* for the low-level piece and *handler* for the high-level piece. Figure 12.4 illustrates the concept.

Figure 12.4 The conceptual division of interrupt code into a dispatcher written in assembly language and a handler written in C.

As the sizes of items in the figure imply, a dispatcher is usually much smaller than a handler. In most systems, a dispatcher consists of a few lines of assembly language code. The dispatcher saves hardware state, usually by pushing copies of the general-purpose registers onto the stack, and calls the handler code. When the handler returns, the dispatcher restores the machine state (e.g., by popping registers off the stack), and then uses the special instruction that returns from the interrupt. Of course, the call to a handler must follow the standard C calling conventions. Therefore, on some architectures, the dispatcher includes a few extra instructions that set up the environment needed by a C function.

12.9.1 Galileo (Intel)

Recall that the Intel interrupt controller stores the interrupt and exception vectors, and invokes code associated with a specific exception or device. For example, when the Ethernet device interrupts, control passes to the Ethernet interrupt code, and when the serial console interrupts, control passes to the code that handles interrupts on the serial console.

Interrupt controller hardware that invokes code for a specific device can save time because the operating system does not need to index interrupt vectors in memory. However, the design has a consequence for device driver software: a driver must include low-level dispatch code. Our Galileo device drivers illustrate the organization: each driver has both a dispatch function (written in assembly language) and a handler function (written in C). Figure 12.5 illustrates the structure.

Figure 12.5 The structure of interrupt code on the Galileo. Each device driver has both a low-level dispatcher and a high-level handler.

12.9.2 BeagleBone Black (ARM)

The interrupt controller hardware used with BeagleBone Black differs from the controller used with the Galileo. The controller on the BeagleBone Black does not store interrupt vectors, nor does it invoke the code for a specific device. Instead, the hardware groups device interrupts into two exceptions: IRQ and FIQ. Because the devices of interest only use the IRQ mechanism, we will ignore FIQ, and assume all devices are grouped into the IRQ exception. When a device interrupts, an IRQ exception occurs, and the hardware invokes a single function in the operating system. A hardware register specifies the IRQ of the interrupting device.

Recall that the exception vectors reside in memory, and the operating system must initialize the vectors at startup. The hardware maps the interrupt from any device into an IRQ exception, jumps to the interrupt vector, which has been initialized with code that will jump to the IRQ dispatcher. The IRQ dispatcher determines which device interrupted, and then calls the appropriate handler. Figure 12.6 illustrates the structure.

Figure 12.6 The two-level interrupt code on the BeagleBone Black. Device exceptions pass to an assembly language function, *irq_except*, which calls a dispatcher written in C, which calls a handler.

Function *irq_dispatch* is found in file *evec.c*. Although the file contains other functions (*initintc* to initialize the interrupt controller hardware and *set_evec* to assign an entry in the parallel exception vector array), we only need to consider *irq_dispatch*. The code appears on the next page.

12.10 Disabling Interrupts

Interrupts play a fundamental role in operating system design. The basic idea is easy to understand. The hardware disables further interrupts once an interrupt occurs. As a result, interrupt code cannot be interrupted. That is, interrupts remain disabled when the dispatcher runs, when the handler is called, and when the handler returns. Interrupts are only enabled when the processor executes a special instruction (or sequence of instructions) that return from the interrupt to the place at which processing was originally interrupted. The point can be summarized:

Interrupts are disabled when an interrupt occurs, and remain disabled until the code returns from an interrupt.

```
/* Snippet of code from evec.c */

/*------------------------------------------------------------------------
 * irq_dispatch  -  Call the handler for specific interrupt
 *------------------------------------------------------------------------
 */
void    irq_dispatch()
{
        struct  intc_csreg *csrptr = (struct intc_csreg *)0x48200000;
        uint32  xnum;                   /* Interrupt number of device   */
        interrupt (*handler)();         /* Pointer to handler function  */

        /* Get the interrupt number from the Interrupt controller */

        xnum = csrptr->sir_irq & 0x7F;

        /* If a handler is set for the interrupt, call it */

        if(intc_vector[xnum]) {
                handler = intc_vector[xnum];
                handler(xnum);
        }

        /* Acknowledge the interrupt */

        csrptr->control |= (INTC_CONTROL_NEWIRQAGR);
}
```

Although it may seem obvious, the interrupt policy stated above has subtle consequences. Device hardware places strict limits on the time a processor can run with interrupts disabled. If an operating system leaves interrupts disabled arbitrarily long, devices will fail to perform correctly. For example, a network device may drop incoming packets if the onboard packet buffer fills and interrupts remain disabled. As a consequence, interrupt routines must be written to complete processing as quickly as possible and resume executing the code that had interrupts enabled. More important, interrupts are global — if the handler for one device leaves interrupts disabled, all devices are affected. Thus, when creating interrupt code, a programmer must be aware of the constraints on other devices in the system, and must accommodate the device with the smallest time constraint. The point can be summarized:

> *The maximum time that an interrupt handler for device* D *can leave interrupts disabled cannot be computed by examining device* D. *Instead, the time is computed by choosing the smallest constraint across all devices in the system.*

The rule may seem obvious and trivial. One must remember, however, that computers are designed to accommodate an arbitrary set of devices and to allow owners to connect new devices at any time. Therefore, when writing an interrupt handler for a specific device, a programmer cannot know the set of devices that will be connected to the computer. Users experience a consequence: a driver for an existing device, *X*, may not be compatible with hardware for a new device, *Y*.

12.11 Constraints On Functions That Interrupt Code Invokes

In addition to ensuring that interrupt code accommodates the device with the tightest time constraints, an operating system designer must build interrupt code to be executed by an arbitrary process. That is, interrupt code is executed by whichever process is running when the interrupt occurs.

The process executing an interrupt seems irrelevant until one considers two facts:

- An interrupt handler can invoke operating system functions.
- Because the scheduler assumes at least one process will remain ready to run, the null process must remain in either the *current* or *ready* state.

The null process is designed to be an infinite loop that does not make function calls. However, an interrupt can be thought of as occurring "between" two successive instructions. Thus, if an interrupt occurs while the null process is executing, the null process will remain running while the handler executes. The most important consequence is:

> *Interrupt routines can only call operating system functions that leave the executing process in the current or ready states.*

That is, interrupt routines may invoke functions such as *send* or *signal*, but may not invoke functions such as *wait* that move the executing process to a non-eligible state.

12.12 The Need To Reschedule During An Interrupt

Consider the question of rescheduling during an interrupt. To see why rescheduling is needed, observe the following:

- The scheduling invariant specifies that at any time, a highest priority eligible process must be executing.
- When an I/O operation completes, a high-priority process may become eligible to execute.

For example, suppose a high-priority process, *P*, chooses to read a packet from the network. Even though it has high priority, *P* must block to wait for a packet. While *P* is blocked, some other process, *Q*, will be running. When a packet arrives and an interrupt occurs, process *Q* will be interrupted, and will execute the interrupt handler. If the interrupt handler merely moves *P* to the ready state and returns, process *Q* will continue to execute. If *Q* has lower-priority than *P*, the scheduling invariant will be violated.

As an extreme case, consider what happens if a system only contains one application process, and the application blocks to wait for I/O. The null process will be running when the interrupt occurs. If the interrupt handler does not reschedule, the interrupt will return to the null process and the application will never execute. The key idea is:

> *To ensure that processes are notified promptly when an I/O operation completes and to maintain the scheduling invariant, an interrupt handler must reschedule whenever it makes a waiting process ready.*

12.13 Rescheduling During An Interrupt

The interaction between the scheduling and interrupt policies creates a complex question: under what circumstances can we allow rescheduling? We said that interrupt routines must keep interrupts disabled while processing an interrupt. We also said that an interrupt handler must re-establish the scheduling invariant if a process that was waiting for I/O becomes eligible to execute. However, consider what can happen during rescheduling. Suppose the process that is selected to execute has been executing with interrupts enabled. Once it begins to execute, the process will return from the scheduler, and interrupts will be allowed. Thus, it might seem that an interrupt handler should not be allowed to reschedule because switching to a process that runs with interrupts enabled could start a cascade of further interrupts. We must convince ourselves that rescheduling during an interrupt is safe as long as global data structures are valid.

To understand why rescheduling is safe, consider the series of events leading to a call of *resched* from an interrupt handler. Suppose a process *U* was running with interrupts enabled when the interrupt occurred. Interrupt dispatching uses *U*'s stack to save the state, and leaves process *U* running with interrupts disabled while executing the interrupt handler. Suppose the handler calls *resched*, which switches to another process, *T*. After *T* returns from the context switch, *T* may be running with interrupts enabled, and another interrupt may occur. What prevents an infinite loop where unfinished interrupts pile up until a stack overflows with interrupt function calls? Recall that each process has its own stack. Process *U* had one interrupt on its stack when it was stopped by the context switch. The new interrupt occurs while the processor is executing process *T*, which means the processor is using *T*'s stack.

Consider process *U*. Before another interrupt can pile up on *U*'s stack, *U* must regain control of the processor and interrupts must be enabled. At the last step before it gave up control, *U* called the scheduler, *resched*, which called the context switch. At

that point, *U* was running with interrupts disabled. Therefore, when *U* regains control (i.e., when the scheduler selects *U* again and performs a context switch), *U* will begin executing with interrupts disabled. That is, *U* will start executing in the context switch. The context switch will return to the scheduler, the scheduler will return to the interrupt handler, and the handler will return to the dispatcher.

During the sequence of returns, *U* will continue to execute with interrupts disabled until the dispatcher returns from the interrupt (i.e., until interrupt processing completes and the dispatcher returns to the location at which the original interrupt occurred). So, no additional interrupts can occur while process *U* is executing interrupt code (even though an interrupt can occur if *U* switches to another process and the other process runs with interrupts enabled). The important constraint is: only one interrupt can be in progress for a given process at any time. Because only a finite number of processes exist in the system at a given time and each process can have at most one outstanding interrupt, the number of outstanding interrupts is bounded. To summarize:

> *Rescheduling during interrupt processing is safe provided that (1) interrupt code leaves global data in a valid state before rescheduling, and (2) no function enables interrupts unless it disabled them.*

The rule explains why all operating systems functions use disable/restore rather than disable/enable. A function that disables interrupts upon entry always restores them before returning to its caller; no routine ever enables interrupts explicitly. The hardware that manages interrupts follows the same paradigm as the operating system. When an interrupt occurs, further interrupts are disabled, and they remain disabled until the processor returns from interrupt processing. The only exception to our rule about disabling and restoring interrupts is found in the system initialization function which enables interrupts explicitly at system startup.†

12.14 Perspective

The relationship between interrupts and processes is among the most subtle and complex aspects of operating systems. Interrupts are low-level mechanisms — they are part of the underlying hardware and are defined in terms of sequential notions such as the fetch-execute cycle. Processes are high-level abstractions — they are imagined by operating system designers and defined by a set of system functions. Consequently, it is easiest to understand interrupts by thinking about the mechanism without thinking about concurrent processes, and easiest to understand concurrent processes by thinking about the abstraction without thinking about interrupts.

Unfortunately, combining the abstract world of processes and the concrete world of interrupts is intellectually challenging. If the interactions between interrupts and processes does not seem incredibly complex, you have not thought about it deeply. If it seems too complex to grasp, console yourself that you are not alone. With careful thought, you will be able to master the basics.

---

†At startup, a processor disables all interrupt processing until the operating system runs and enables them.

12.15 Summary

To process an interrupt, the hardware and operating system cooperate to save a copy of the processor state, determine which device requested the interrupt, and invoke a handler for the device. Because a high-level language does not provide facilities to manipulate machine state, interrupt code is divided into a dispatcher that is written in assembly language and a handler that is written in a high-level language, such as C. On the Galileo, a separate dispatch function is needed for each device; on the BeagleBone Black, a single dispatch function handles all devices.

Three basic rules control interrupt processing. First, interrupt code must not leave interrupts disabled arbitrarily long or devices will fail to operate correctly. The length of time an interrupt can be delayed depends on all devices attached to the system, not only on the device being serviced. Second, because it can be executed by the null process, interrupt code must never call a function that will move the executing process out of the *current* or *ready* states. Third, an interrupt handler must use a special instruction to return from an interrupt; the code must never enable interrupts explicitly.

Despite the prohibition on enabling interrupts, a handler can call *resched* whenever a waiting process becomes ready. Doing so re-establishes the scheduling invariant and also means that if a process is waiting for I/O to complete, the process will be informed promptly. Of course, the code must ensure that global data structures are in a valid state before rescheduling. Rescheduling does not cause a cascade of interrupts because each process can have at most one interrupt on its stack.

EXERCISES

12.1 Suppose an interrupt handler contains an error that explicitly enables interrupts. Describe how a system might fail.

12.2 Modify interrupt handlers to enable interrupts, and see how long a system can run before crashing. Are you surprised? Determine *exactly* why the system crashes. (Note: for this exercise, disable the timer device that is described in the next chapter.)

12.3 On the BeagleBone Black platform, how many functions are called each time a character arrives over a serial line? Can the number of calls be reduced, or does the software architecture depend on the hardware? Explain.

12.4 Imagine a processor where the hardware automatically switches context to a special "interrupt process" whenever an interrupt occurs. The only purpose of the interrupt process is to run interrupt code. Does such a design make an operating system easier or more difficult to design? Explain. Hint: will the interrupted process be permitted to reschedule?

12.5 If you could redesign interrupt controller hardware, what changes would you make to minimize the work the operating system needs to do?

12.6 Calculate how many microseconds can be spent per interrupt assuming eight serial devices each receive characters at 115 Kbaud (115 thousand bits per second, or approximately 11,500 characters per second).

12.7 Download a copy of the Xinu source code, and examine the interrupt functions in *intr.S* for both the Galileo and BeagleBone Black (for the Galileo, also include the dispatch function from a device driver). Explain the purpose of each line of code.

12.8 Read about IRQ mode in the ARM architecture. Which registers are banked? Why?

12.9 Download a copy of source code for both the ARM and Galileo versions of Xinu. Compare function *set_evec* for the two architectures. What is the chief difference?

Chapter Contents

13.1 Introduction, 233
13.2 Timed Events, 234
13.3 Real-time Clocks And Timer Hardware, 234
13.4 Handling Real-time Clock Interrupts, 235
13.5 Delay And Preemption, 236
13.6 Implementation Of Preemption, 237
13.7 Efficient Management Of Delay With A Delta List, 238
13.8 Delta List Implementation, 239
13.9 Putting A Process To Sleep, 241
13.10 Timed Message Reception, 244
13.11 Awakening Sleeping Processes, 248
13.12 Clock Interrupt Processing, 249
13.13 Clock Initialization, 251
13.14 Perspective, 254
13.15 Summary, 255

13

Real-time Clock Management

We haven't the time to take our time.

— Eugene Ionesco

13.1 Introduction

Earlier chapters describe two major pieces of an operating system: a process manager that provides concurrent processing and a memory manager that allows blocks of memory to be allocated and released dynamically. The previous chapter introduces interrupt processing. The chapter states rules for interrupt processing, describes how the operating system captures control when an interrupt occurs, and explains how control passes through a dispatcher to a device-specific interrupt handler.

This chapter continues the discussion of interrupts by describing timing hardware and explaining how an operating system uses a real-time mechanism to provide processes with the ability to control timed events. The chapter introduces two fundamental concepts: a delta list data structure and process preemption. It explains how an operating system uses a clock to provide round-robin service to a set of equal-priority processes. Later chapters extend the study of interrupts by exploring device drivers for other I/O devices.

13.2 Timed Events

Many applications use *timed events*. For example, an application might create a window to display a message, leave the window on the screen for five seconds, and then remove the window. An application that prompts for a password might choose to exit unless a password is entered within thirty seconds. Parts of an operating system also use timed events. For example, many network protocols require a sender to retransmit a request if no response is received within a specified time. Similarly, an operating system might choose to inform a user if a peripheral, such as a printer, remains disconnected for more than a few seconds. On small embedded systems that do not have a separate hardware mechanism, an operating system uses timed events to maintain the current date and time of day.

Because time is fundamental, most operating systems provide facilities that make it easy for an application to create and manage a set of timed events. Some systems use a general-purpose *asynchronous event paradigm* in which a programmer defines a set of event handlers and the operating system invokes the appropriate handler when an event occurs. Timed events fit into the asynchronous paradigm easily: a running process requests that a specific event occur T time units in the future. Other systems follow a *synchronous event paradigm* in which the operating system only provides delay and a programmer creates extra processes as needed to schedule events. Our example system uses the synchronous approach.

13.3 Real-time Clocks And Timer Hardware

Four types of hardware devices relate to time:

- Processor clock
- Real-time clock
- Time-of-day clock
- Interval timer

Processor clock. The term *processor clock* refers to a hardware device that emits pulses (i.e., square waves) at regular intervals with high precision. The processor clock controls the rate at which the processor executes instructions. To minimize hardware, low-end embedded systems often use the processor clock as a source of timing information. Unfortunately, a processor clock rate is often inconvenient (i.e., the clock pulses rapidly, and the rate may not divide evenly into one second).

Real-time clock. A real-time clock operates independent of the processor, and pulses in fractions of a second (e.g., 1000 times per second), generating an interrupt for each pulse. Usually, real-time clock hardware does not count pulses — if an operating system needs to compute an elapsed time, the system must increment a counter when each clock interrupt occurs.

Time-of-day clock. Technically, a time-of-day clock is a chronometer that computes elapsed time. The hardware consists of an internal real-time clock connected to a counter that tallies the pulses. Like a normal clock, the time can be changed. Once set, however, the mechanism runs independent of the processor, and continues as long as the system receives power (some units include a small battery to keep the clock active even if the external power is removed temporarily). Unlike other clocks, a time-of-day clock does not interrupt — the processor must set or interrogate the clock.

Interval timer. An interval timer, sometimes called a *count-down timer* or simply a *timer*, also consists of an internal real-time clock and a counter. To use an interval timer, the system initializes the counter to a positive value. The timer decrements the count once for each real-time clock pulse, and generates an interrupt when the count reaches zero. A variant known as a *count-up timer* requires the operating system to initialize the count to zero and set a *limit*. As the name implies, a count-up timer increments the counter, and interrupts the operating system when the counter reaches the limit value.

The chief advantage of a timer over a real-time clock lies in lower interrupt overhead. A real-time clock interrupts regularly, even if the next event is many time units in the future. A timer only interrupts when an event is scheduled. Furthermore, a timer is more flexible than a real-time clock because a timer can emulate a real-time clock. To emulate a real-time clock with a rate of R pulses per second, for example, a timer is set to interrupt in $1/R$ seconds. When an interrupt occurs, the timer is reset to the same value. To summarize:

> *The hardware an operating system uses to manage timed events consists of real-time clocks and interval timers. A real-time clock interrupts regularly; an interval timer interrupts after a specified delay.*

In terms of the example platforms, the Galileo has a real-time clock, and the BeagleBone Black has an interval timer. At startup, the Galileo code configures the real-time clock to interrupt once every millisecond. The BeagleBone Black code configures the interval timer to act like a real-time clock by specifying that the timer should interrupt after a millisecond and should automatically restart when an interrupt occurs. Thus, the following sections that describe how an operating system uses a real-time clock apply to both of the example platforms.

13.4 Handling Real-time Clock Interrupts

We said that a real-time clock interrupts regularly without counting or accumulating interrupts. Similarly, if a timer is used to emulate a real-time clock, the timer does not accumulate interrupts. In either case, if a processor fails to service a clock interrupt before the clock pulses again, the processor will not receive the second interrupt. More important, the hardware does not detect or report the error†:

†Although hardware exists that can report overflow, many clock modules do not.

If a processor takes too long to service a real-time clock interrupt or if it operates with interrupts disabled for more than one clock cycle, a clock interrupt will be missed and no error will be reported.

The operation of real-time clock hardware has two significant consequences for system designers. First, because it must be able to execute many instructions between real-time clock interrupts, a processor must operate significantly faster than the real-time clock. Second, real-time clock interrupts can be a source of hidden errors. That is, if an operating system runs too long with interrupts disabled, clock interrupts will be missed and timing will be affected. Such errors can easily go undetected.

Obviously, systems must be designed to service clock interrupts quickly. Some hardware helps by giving highest priority to real-time clock interrupts. Thus, if an I/O device and a clock device each request an interrupt at the same time, the processor receives the clock interrupt first, and only receives the I/O interrupt after the clock has been serviced.

13.5 Delay And Preemption

We will focus on two ways that an operating system uses timed events:

- Timed delay
- Preemption

Timed delay. An operating system allows any process to request a timed delay. When a process requests a timed delay, the operating system moves the process from the current state into a new state (which we call *sleeping*), and schedules a *wakeup* event to restart the process at the specified time. When the wakeup event occurs, the process becomes eligible to use the processor, and executes according to the scheduling policy. Later sections explain how a process is put to sleep and how it is reawakened at the correct time.

Preemption. The process manager in an operating system uses a preemption mechanism to implement *time slicing* that guarantees equal-priority processes receive service round-robin, as specified by the scheduling policy in Chapter 5. The system defines a maximum time slice, T, that a process can execute without allowing other processes to execute. When it switches from one process to another, the scheduler schedules a preemption event T time units in the future. When a preemption event occurs, the event handler simply calls *resched*.

To understand how preemption works, observe that a system may contain multiple processes with the same priority. Thus, while one process executes, other processes of equal priority may be enqueued on the ready list, eligible to run. In such cases, a call to *resched* places the current process at the end of the ready list, behind other processes with equal priority, and switches to the first process on the list. Therefore, if k equal-

priority processes are ready to use the processor, all *k* execute for at most one time slice before any process receives more service.

How long should a time slice be? We say that the choice of a time slice controls the *granularity of preemption*. Using a short time slice makes the granularity small by rescheduling often. Small granularity tends to keep all equal priority processes proceeding at approximately the same pace because no process can run for more than *T* time units before another has an opportunity to run. However, a small granularity introduces higher overhead because the system switches context often. If the granularity is too small, the system will spend more time handling clock interrupts and context switching than executing application processes. A large granularity reduces the overhead of context switching, but allows a process to hold the processor longer before allowing other processes to execute.

It turns out that in most systems, a process seldom uses the processor long enough for preemption to occur. Instead, a process usually performs I/O or executes a system function, such as *wait*, that causes rescheduling. In essence, a process voluntarily gives up control of the processor before its timeslice ends. More important, because input and output are slow compared to processing, processes spend most of their time waiting for I/O to complete. Despite the expected case, preemption provides important functionality:

> *Without a preemptive capability, an operating system cannot regain control from a process that executes an infinite loop.*

13.6 Implementation Of Preemption

The example code implements both preemption and timed delays; before examining the code, we will discuss each. Preemption is the easiest to understand. Defined constant *QUANTUM* specifies the number of clock ticks in a single time slice. Whenever it switches from one process to another, *resched* sets global variable *preempt* to *QUANTUM*. Each time the clock ticks, the clock interrupt handler decrements *preempt*. When *preempt* reaches zero, the clock interrupt handler resets *preempt* to *QUANTUM* and calls *resched*. Following the call to *resched*, the handler returns from the interrupt.

The call to *resched* has two possible outcomes. First, if the currently executing process remains the only process at the highest priority, *resched* will return immediately, the interrupt handler will return, and the current process will return to the point of the interrupt and continue executing for another timeslice. Second, if another ready process has the same priority as the current process, the call to *resched* will switch to the new process. Eventually, *resched* will switch back to the interrupted process. The assignment of *QUANTUM* to *preempt* handles the case where the current process remains running. The assignment is needed because *resched* only resets the preemption counter when it switches to a new process.† Note that resetting the preemption counter prevents the counter from underflowing in cases where a single process executes indefinitely.

†The code for *resched* can be found on page 82.

13.7 Efficient Management Of Delay With A Delta List

To implement delay, the operating system must maintain information about the set of processes that have requested a delay. Each process specifies a delay relative to the time at which it places a request, and additional processes can make a request at any time. When the delay for a process expires, the system makes the process ready and calls *resched*.

How can an operating system maintain a set of processes that have each requested a specific delay? The system cannot afford to search through arbitrarily long lists of sleeping processes on each clock tick. Therefore, an efficient data structure is needed that only requires a clock interrupt handler to execute a few instructions on each clock tick while accommodating a set of processes that have each requested a specific delay.

The solution lies in a data structure that we call a *delta list*. A delta list contains a set of processes, and the list is ordered by the time at which a process should awaken. The fundamental insight needed to make computation efficient lies in the use of *relative* rather than *absolute* times. That is, instead of storing a value that specifies the time a process should awaken, a key in the delta list stores the additional time a process must delay beyond the preceding process on the list:

> *The key of the first process on a delta list specifies the number of clock ticks a process must delay beyond the current time; the key of each other process on a delta list specifies the number of clock ticks the process must delay beyond the preceding process on the list.*

As an example, suppose processes A, B, C, and D request delays of 6, 12, 27, and 50 ticks, respectively. Further suppose the requests are made at approximately the same time (i.e., within one clock tick). Figure 13.1 illustrates the delta list that will result.

Figure 13.1 Conceptual form of a delta list with four processes that have delays 6, 12, 27, and 50 ticks from the current time, respectively.

Given a delta list, one can find the time at which each process will awaken by computing partial sums of keys. In the figure, the delay before process *A* awakens is 6, the delay before process *B* awakens is $6+6$, the delay before process *C* awakens is $6+6+15$, and the delay before *D* awakens is $6+6+15+23$.

13.8 Delta List Implementation

An overarching goal of operating systems designers arises from the desire to achieve maximal functionality with minimal mechanism. Designs that provide powerful functionality with minimal overhead are valued. To achieve such goals, designers look for ways to create underlying mechanisms that accommodate multiple functions. In the case of delta lists, we will see that it is possible to use the basic list data structure covered in Chapter 4. That is, the delta list of delayed processes will reside in the *queuetab* structure, just like other lists of processes.

Conceptually, the processing required for a delta list is straightforward. Global variable *sleepq* contains the queue ID of the delta list for sleeping processes. On each clock tick, the clock interrupt handler examines the queue of sleeping processes, and decrements the key on the first item if the queue is nonempty. If the key reaches zero, the delay has expired and the process must be awakened. To awaken a process, the clock handler calls function *wakeup*

Functions to manipulate a delta list seem straightforward, but the implementation can be tricky. Therefore, a programmer must pay close attention to details. Function *insertd* takes three arguments: a process ID, *pid*, a queue ID, *q*, and a delay given by argument *key*. *Insertd* finds the location on the delta list where the new process should be inserted and links the process into the list. In the code, variable *next* scans the delta list searching for the place to insert the new process. File *insertd.c* contains the code.

Observe that the initial value of argument *key* specifies a delay relative to the current time. Thus, argument *key* can be compared to the key in the first item on the delta list. However, successive keys in the delta list specify delays relative to their predecessor. Thus, the key in successive nodes on the list cannot be compared directly to the value of argument *key*. To keep the delays comparable, *insertd* subtracts the relative delays from *key* as the search proceeds, maintaining the following invariant:

> *At any time during the search, both key and queuetab[next].qkey specify a delay relative to the time at which the predecessor of "next" awakens.*

Although *insertd* checks for the tail of the list explicitly during the search, the test could be removed without affecting the execution. To understand why, recall that the key value in the tail of a list is assumed to be greater than any key being inserted. As long as the assertion holds, the loop will terminate once the tail has been reached. Because *insertd* does not check its argument, keeping the test provides a safety check.

After it has identified a location on the list where the relative delay of the item being inserted is smaller than the relative delay of an item on the list, *insertd* links the new item into the list. *Insertd* must also subtract the extra delay that the new item introduces from the delay of the rest of the list. To do so, *insertd* decrements the key in the next item on the list by the key value being inserted. The subtraction is guaranteed

to produce a nonnegative value because the termination condition for the loop guarantees that the key inserted is less than the next key on the list.

```
/* insertd.c - insertd */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  insertd  -  Insert a process in delta list using delay as the key
 *------------------------------------------------------------------------
 */
status  insertd(                        /* Assumes interrupts disabled  */
          pid32         pid,            /* ID of process to insert      */
          qid16         q,              /* ID of queue to use           */
          int32         key             /* Delay from "now" (in ms.)    */
        )
{
        int32   next;                   /* Runs through the delta list  */
        int32   prev;                   /* Follows next through the list*/

        if (isbadqid(q) || isbadpid(pid)) {
                return SYSERR;
        }

        prev = queuehead(q);
        next = queuetab[queuehead(q)].qnext;
        while ((next != queuetail(q)) && (queuetab[next].qkey <= key)) {
                key -= queuetab[next].qkey;
                prev = next;
                next = queuetab[next].qnext;
        }

        /* Insert new node between prev and next nodes */

        queuetab[pid].qnext = next;
        queuetab[pid].qprev = prev;
        queuetab[pid].qkey = key;
        queuetab[prev].qnext = pid;
        queuetab[next].qprev = pid;
        if (next != queuetail(q)) {
                queuetab[next].qkey -= key;
        }

        return OK;
}
```

13.9 Putting A Process To Sleep

An application does not call *insertd*, nor does the application access the sleep queue directly. Instead, an application invokes system call *sleep* or *sleepms* to request a delay. The only difference between the two functions is the granularity of their arguments. An argument to *sleepms* specifies a delay in milliseconds, the smallest granularity delay that is possible when a clock interrupts every millisecond. An argument to *sleep* specifies a delay in seconds, which is easier to use in some cases. For example, a delay visible to a human is often expressed in seconds rather than milliseconds.

To avoid duplicating code, function *sleep* multiplies its argument by 1000 and invokes *sleepms*. The only interesting aspect of *sleep* is a check on its argument size: to avoid integer overflow, *sleep* limits the delay to a value that can be represented as a 32-bit unsigned integer. If the caller specifies a larger value, *sleep* returns *SYSERR*.

On a 32-bit processor, measuring delay in milliseconds provides an adequate range of delay for most applications. An unsigned 32-bit integer accommodates delays over 1100 hours (49 days). Delays longer than 49 days can be managed by having a process repeatedly sleep for many days, awaken, check the time, and sleep again. On embedded systems that use 16-bit integers, however, millisecond delays mean that a caller can only express a delay of thirty-two seconds. Such systems seldom have much memory or processing power, so using a process to manage longer delays may not be feasible. Therefore, an operating system designed for a slow, 16-bit processor may choose a larger granularity for clock interrupts (e.g., tenths of seconds instead of milliseconds). If the clock generates interrupts every tenth of a second, a sleep function must be changed to measure delays in tenths of seconds.

The choice of delay granularity may also be limited by the speed of the processor. Handling clock interrupts can take a surprising amount of time because they never stop, even when no processes are sleeping. If a clock interrupts too fast, a processor will spend most of its time handling clock interrupts. Fortunately, processors have become extremely fast. As processor speeds increased, it became possible to increase the rate of clock interrupts, allowing the granularity of delays to decrease. Thus, the fastest processors allow microsecond delays.

Consider the state of a sleeping process. We said that to delay a process, *sleepms* inserts the process into the delta list of sleeping processes. When it has been moved to the list of sleeping processes, the process is no longer *ready* or *current*. In what state should it be placed? Sleeping differs from suspension, waiting to receive a message, or waiting for a semaphore. Thus, because none of the existing states suffices, a new process state must be added to the design. We call the new state *sleeping*, and denote it with symbolic constant *PR_SLEEP*. Figure 13.2 illustrates state transitions that include the sleeping state.

Figure 13.2 State transitions including the *sleeping* state.

The implementation of *sleepms*, shown below in file *sleep.c*, includes a special case: if a process specified a delay of zero, *sleepms* does not delay the process, but calls *resched* immediately. Otherwise, *sleepms* uses *insertd* to insert the current process in the delta list of sleeping processes, changes the state to *sleeping*, and calls *resched* to allow other processes to execute.

```
/* sleep.c - sleep sleepms */

#include <xinu.h>

#define MAXSECONDS      4294967         /* Max seconds per 32-bit msec  */
```

```
/*------------------------------------------------------------------------
 *  sleep  -  Delay the calling process n seconds
 *------------------------------------------------------------------------
 */
syscall sleep(
          uint32        delay           /* Time to delay in seconds     */
        )
{
        if (delay > MAXSECONDS) {
                return SYSERR;
        }
        sleepms(1000*delay);
        return OK;
}

/*------------------------------------------------------------------------
 *  sleepms  -  Delay the calling process n milliseconds
 *------------------------------------------------------------------------
 */
syscall sleepms(
          uint32        delay           /* Time to delay in msec.       */
        )
{
        intmask mask;                   /* Saved interrupt mask         */

        mask = disable();
        if (delay == 0) {
                yield();
                restore(mask);
                return OK;
        }

        /* Delay calling process */

        if (insertd(currpid, sleepq, delay) == SYSERR) {
                restore(mask);
                return SYSERR;
        }

        proctab[currpid].prstate = PR_SLEEP;
        resched();
        restore(mask);
        return OK;
}
```

13.10 Timed Message Reception

Xinu includes a mechanism that is especially useful in computer networking: timed message reception. In essence, the mechanism allows a process to wait for a specified time or for a message to arrive, whichever occurs first. The mechanism enhances the *send* and *receive* functions. That is, the mechanism operates like the synchronous *receive* function with an additional provision that places a bound on the maximum time the process will wait.

The fundamental concept behind timed message reception is *disjunctive wait*: a process blocks until one of two events occurs. Many network protocols use disjunctive wait to implement timeout-and-retransmission, a technique senders employ to handle packet loss. When it sends a message, a sender also starts a timer, and then waits for a reply to arrive or the timer to expire, whichever occurs first. If a reply arrives, the network cancels the timer. If the message or the reply is lost, the timer expires, and the protocol software retransmits a copy of the request.

In Xinu, when a process requests a timed receive, the process is placed on the queue of sleeping processes, exactly like any other sleeping process. Instead of assigning the process state *PR_SLEEP*, however, the system places the process in state *PR_RECTIM* to indicate that it is engaged in a receive with timeout. If the sleep timer expires, the process is awakened like any other sleeping process. If a message arrives before the delay expires, the process must be removed from the sleeping process queue. In our implementation, *send* handles the task of removing a process. That is, when sending a message to a process that has performed a disjunctive wait, *send* calls *unsleep* to remove the process from the queue of sleeping processes, and then proceeds to deliver the message.

How can a process know whether the timer expired before a message arrived? In our implementation, the process tests the presence of a message. That is, once it resumes execution after a timed delay, the process checks its process table entry to see if a message has arrived. If no message is present, the timer must have expired. An exercise explores the consequences of the implementation.

Figure 13.3 shows the state diagram with a new state, *TIMED-RECV*, for timed message reception.

Figure 13.3 State transitions including the *timed receive* state.

As we have seen, the *send*† function in Chapter 8 handles the case where a process is in the timed receive state. Thus, we only need to examine the code for function *recvtime* and *unsleep*. Function *recvtime* is almost identical to function *receive*‡ except that before calling *resched*, *recvtime* calls *insertd* to insert the calling process on the queue of sleeping processes and assigns state *PR_RECTIM* instead of state *PR_RECV*. File *recvtime.c* contains the code.

†Function *send* can be found in file *send.c* on page 147.
‡Function *receive* can be found in file *receive.c* on page 148.

```
/* recvtime.c - recvtime */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  recvtime  -  Wait specified time to receive a message and return
 *------------------------------------------------------------------------
 */
umsg32  recvtime(
          int32         maxwait         /* Ticks to wait before timeout */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  procent *prptr;         /* Tbl entry of current process */
        umsg32  msg;                    /* Message to return            */

        if (maxwait < 0) {
                return SYSERR;
        }
        mask = disable();

        /* Schedule wakeup and place process in timed-receive state */

        prptr = &proctab[currpid];
        if (prptr->prhasmsg == FALSE) { /* If message waiting, no delay */
                if (insertd(currpid,sleepq,maxwait) == SYSERR) {
                        restore(mask);
                        return SYSERR;
                }
                prptr->prstate = PR_RECTIM;
                resched();
        }

        /* Either message arrived or timer expired */

        if (prptr->prhasmsg) {
                msg = prptr->prmsg;     /* Retrieve message             */
                prptr->prhasmsg = FALSE;/* Reset message indicator      */
        } else {
                msg = TIMEOUT;
        }
        restore(mask);
        return msg;
}
```

Function *unsleep* is an internal function that removes a process from the queue of sleeping processes.† To maintain the correct delays for processes that remain in the queue, *unsleep* must keep all delays the same as before the deletion. To so do, *unsleep* examines the successor of the process being removed. If a successor exists, *unsleep* adds the delay of the process being removed to the delay of the successor. File *unsleep.c* contains the code.

```
/* unsleep.c - unsleep */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  unsleep  -  Internal function to remove a process from the sleep
 *                  queue prematurely.  The caller must adjust the delay
 *                  of successive processes.
 *------------------------------------------------------------------------
 */
status  unsleep(
          pid32         pid             /* ID of process to remove      */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  procent *prptr;         /* Ptr to process' table entry  */

        pid32   pidnext;                /* ID of process on sleep queue */
                                        /*   that follows the process   */
                                        /*   which is being removed     */

        mask = disable();

        if (isbadpid(pid)) {
                restore(mask);
                return SYSERR;
        }

        /* Verify that candidate process is on the sleep queue */

        prptr = &proctab[pid];
        if ((prptr->prstate!=PR_SLEEP) && (prptr->prstate!=PR_RECTIM)) {
                restore(mask);
                return SYSERR;
        }

        /* Increment delay of next process if such a process exists */
```

†The term *internal* emphasizes that *unsleep* is not a system call, and should not be called by a user process.

```
        pidnext = queuetab[pid].qnext;
        if (pidnext < NPROC) {
                queuetab[pidnext].qkey += queuetab[pid].qkey;
        }

        getitem(pid);                   /* Unlink process from queue */
        restore(mask);
        return OK;
}
```

13.11 Awakening Sleeping Processes

We said that on each tick of the clock, the clock interrupt handler decrements the count of the first key on *sleepq*, and calls *wakeup* to awaken the process when the delay reaches zero. In fact, *wakeup* does not merely awaken one process — it must handle the case where multiple processes are scheduled to awaken at the same time. Therefore, after deferring rescheduling, *wakeup* iterates through all processes that have a delay of zero, using *sleepq* to remove the process from the sleep queue and *ready* to make the process eligible for processor service. (Note that because it has been called from an interrupt dispatcher, which has interrupts disabled, *wakeup* can call *ready* without explicitly disabling interrupts.) Once it finishes moving processes to the ready list, *wakeup* calls *resched_cntl* to allow rescheduling, which means that a context switch will occur if a higher-priority process became ready.

```
/* wakeup.c - wakeup */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  wakeup  -  Called by clock interrupt handler to awaken processes
 *------------------------------------------------------------------------
 */
void    wakeup(void)
{
        /* Awaken all processes that have no more time to sleep */

        resched_cntl(DEFER_START);
        while (nonempty(sleepq) && (firstkey(sleepq) <= 0)) {
                ready(dequeue(sleepq));
        }

        resched_cntl(DEFER_STOP);
        return;
}
```

13.12 Clock Interrupt Processing

We are now ready to examine the code that handles clock interrupts. The clock is configured to interrupt once each millisecond. Recall that interrupt processing is arranged so the interrupt mechanism invokes a dispatch function written in assembly language that saves processor state and invokes a handler that is written in C. On the Galileo, each device has a separate dispatch function; the clock dispatch function is *clkdisp*. The file only contains a few lines of code that save registers, call the interrupt handler, *clkhandler*, restore registers, and return from the interrupt.

```
/* clkdisp.s - clkdisp (x86) */

/*------------------------------------------------------------------------
 * clkdisp  -  Interrupt dispatcher for clock interrupts (x86 version)
 *------------------------------------------------------------------------
 */
#include <icu.s>
                .text
                .globl  clkdisp         # Clock interrupt dispatcher
clkdisp:
                pushal                  # Save registers
                cli                     # Disable further interrupts
                movb    $EOI,%al        # Reset interrupt
                outb    %al,$OCW1_2

                call    clkhandler      # Call high level handler

                sti                     # Restore interrupt status
                popal                   # Restore registers
                iret                    # Return from interrupt
```

The clock interrupt handler, *clkhandler*, which manages sleeping processes and preemption, can be found in file *clkhandler.c*.

```
/* clkhandler.c - clkhandler */

#include <xinu.h>

/*------------------------------------------------------------------------
 * clkhandler - high level clock interrupt handler
 *------------------------------------------------------------------------
 */
void    clkhandler()
{
        static  uint32  count1000 = 1000;       /* Count to 1000 ms     */

        /* Decrement the ms counter, and see if a second has passed */

        if((--count1000) <= 0) {

                /* One second has passed, so increment seconds count */

                clktime++;

                /* Reset the local ms counter for the next second */

                count1000 = 1000;
        }

        /* Handle sleeping processes if any exist */

        if(!isempty(sleepq)) {

                /* Decrement the delay for the first process on the     */
                /*   sleep queue, and awaken if the count reaches zero  */

                if((--queuetab[firstid(sleepq)].qkey) <= 0) {
                        wakeup();
                }
        }

        /* Decrement the preemption counter, and reschedule when the */
        /*   remaining time reaches zero                             */

        if((--preempt) <= 0) {
                preempt = QUANTUM;
                resched();
        }
}
```

Clkhandler begins by incrementing local variable *count1000*, which counts from 1000 down to 0. When *count1000* reaches zero, one second (i.e., 1000 ms.) has elapsed, so *clkhandler* increments global variable *clktime*, which stores the time in seconds since the system booted. *Clktime* is used to provide the date (e.g., it is used by the Xinu shell command *date*).

Once it has handled incrementing global counters, *clkhandler* performs two tasks related to processes: sleeping processes and time slicing. To manage sleeping processes, *clkhandler* decrements the time remaining on the first process in *sleepq* (provided *sleepq* is nonempty). If the remaining delay reaches zero, *clkdisp* calls *wakeup*, which removes all processes from the sleep queue that have a zero delay. As we have seen, *wakeup* makes the processes *ready*. Finally, *clkint* decrements the preemption counter, calling *resched* if the preemption counter reaches zero.

13.13 Clock Initialization

Clock initialization can be divided into two conceptual parts: initialization related to the operating system and initialization related to the underlying clock hardware. In terms of the operating system, the clock initialization code performs three steps. First, it allocates a queue to hold the delta list of sleeping processes, and stores the queue ID in global variable *sleepq*. Second, it initializes the preemption counter, *preempt* to *QUANTUM*. Third, the code initializes global variable *clktime*, which gives the seconds since the system booted, to zero.

In terms of clock hardware initialization, the details vary widely among platforms. On the simplest systems, the clock hardware is completely preconfigured — both the interrupt vector and clock rate are hardwired. On most systems, the hardware is parameterized, which means the operating system can control the rate at which interrupts are generated. The operating system, may also be able to assign an interrupt vector.

Both the Galileo and BeagleBone Black platforms have configurable clock hardware. On the Galileo, the operating system must assign an interrupt vector, set the mode of the clock, and set the exact clock rate. Our code calls *set_evec* to configure *clkdisp* to be the function that receives interrupts. It then uses an *outb* call to configure the clock as a 16-bit timer. Finally, our code uses two calls of *outb* to configure the 16-bit rate register to 1193, which will cause the clock to interrupt once per millisecond. The clock initialization code for the Galileo can be found in file *clkinit.c*.

```
/* clkinit.c - clkinit (x86) */

#include <xinu.h>

uint32  clktime;                /* Seconds since boot                   */
uint32  ctr1000 = 0;            /* Milliseconds since boot              */
qid16   sleepq;                 /* Queue of sleeping processes          */
uint32  preempt;                /* Preemption counter                   */

/*------------------------------------------------------------------------
 * clkinit  -  Initialize the clock and sleep queue at startup (x86)
 *------------------------------------------------------------------------
 */
void    clkinit(void)
{
        uint16  intv;           /* Clock rate in KHz                    */

        /* Allocate a queue to hold the delta list of sleeping processes*/

        sleepq = newqueue();

        /* Initialize the preemption count */

        preempt = QUANTUM;

        /* Initialize the time since boot to zero */

        clktime = 0;

        /* Set interrupt vector for the clock to invoke clkdisp */

        set_evec(IRQBASE, (uint32)clkdisp);

        /* Set the hardware clock: timer 0, 16-bit counter, rate */
        /*   generator mode, and counter runs in binary          */

        outb(CLKCNTL, 0x34);

        /* Set the clock rate to 1.190 Mhz; this is 1 ms interrupt rate */

        intv = 1193;    /* Using 1193 instead of 1190 to fix clock skew */

        /* Must write LSB first, then MSB */

        outb(CLOCK0, (char) (0xff & intv) );
```

```
        outb(CLOCK0, (char) (0xff & (intv>>8)));

        return;
}
```

The clock initialization code for the BeagleBone Black is also in a file named *clkinit.c*; the first line of the file indicates the platform.

```
/* clkinit.c - clkinit (BeagleBone Black) */

#include <xinu.h>

uint32  clktime;                /* Seconds since boot                   */
uint32  ctr1000 = 0;            /* Milliseconds since boot              */
qid16   sleepq;                 /* Queue of sleeping processes          */
uint32  preempt;                /* Preemption counter                   */

/*------------------------------------------------------------------------
 * clkinit  -  Initialize the clock and sleep queue at startup
 *------------------------------------------------------------------------
 */
void    clkinit(void)
{
        volatile struct am335x_timer1ms *csrptr =
        (volatile struct am335x_timer1ms *)AM335X_TIMER1MS_ADDR;
                        /* Pointer to timer CSR in BBoneBlack   */
        volatile uint32 *clkctrl =
                        (volatile uint32 *)AM335X_TIMER1MS_CLKCTRL_ADDR;

        *clkctrl = AM335X_TIMER1MS_CLKCTRL_EN;
        while((*clkctrl) != 0x2) /* Do nothing */ ;

        /* Reset the timer module */

        csrptr->tiocp_cfg |= AM335X_TIMER1MS_TIOCP_CFG_SOFTRESET;

        /* Wait until the reset os complete */

        while((csrptr->tistat & AM335X_TIMER1MS_TISTAT_RESETDONE) == 0)
                /* Do nothing */ ;

        /* Set interrupt vector for clock to invoke clkint */

        set_evec(AM335X_TIMER1MS_IRQ, (uint32)clkhandler);
```

```
        sleepq = newqueue();    /* Allocate a queue to hold the delta   */
                                /*   list of sleeping processes         */

        preempt = QUANTUM;      /* Set the preemption time              */

        clktime = 0;            /* Start counting seconds               */

        /* The following values are calculated for a    */
        /*   timer that generates 1ms tick rate         */

        csrptr->tpir = 1000000;
        csrptr->tnir = 0;
        csrptr->tldr = 0xFFFFFFFF - 26000;

        /* Set the timer to auto reload */

        csrptr->tclr = AM335X_TIMER1MS_TCLR_AR;

        /* Start the timer */

        csrptr->tclr |= AM335X_TIMER1MS_TCLR_ST;

        /* Enable overflow interrupt which will generate */
        /*   an interrupt every 1 ms                     */

        csrptr->tier = AM335X_TIMER1MS_TIER_OVF_IT_ENA;

        /* Kickstart the timer */

        csrptr->ttgr = 1;

        return;
}
```

13.14 Perspective

Clock and timer management are both technically and intellectually challenging. On the one hand, because clock or timer interrupts occur frequently and have high priority, the total time a processor spends executing clock interrupts is large and other interrupts are prevented. Thus, the code for an interrupt handler must be written carefully to minimize the time taken to handle a given interrupt. On the other hand, an operating system that allows processes to request timed events can schedule many events to occur at exactly the same time, which means that the time taken for a given

interrupt can be arbitrarily long. The conflict can become especially important in real-time embedded systems where the processor is relatively slow and other devices need prompt service.

Most systems allow arbitrary events to be scheduled and defer processing when multiple events collide — a cell phone may not update the display exactly when an application starts, or a text message may take longer to deliver when multiple applications are running. The intellectual questions are: how can an operating system best provide the illusion of precise timing within hardware constraints and inform users when requests cannot be serviced? Should events be assigned priorities? If so, how should event priorities interact with scheduling priorities? There are no easy answers.

13.15 Summary

A real-time clock interrupts the processor at regular intervals. An operating system uses the clock to handle preemption and process delay. A preemption event, scheduled every time the system switches context, forces a call to the scheduler after a process has used the processor for *QUANTUM* clock ticks. Preemption guarantees that no process uses the processor forever, and enforces the scheduling policy by ensuring round-robin service among equal-priority processes.

A delta list provides an efficient way to manage sleeping processes. When a process requests a timed delay (i.e., sleeps), the process places itself on the delta list of sleeping processes. Later, when its delay expires, the clock interrupt handler awakens the sleeping process by moving the process back to the ready list and rescheduling.

EXERCISES

13.1 Modify the code to generate clock interrupts ten times faster, and arrange for the clock interrupt handler to ignore nine interrupts before processing one. How much extra overhead do the additional interrupts generate?

13.2 Trace the series of function calls that occur starting with a clock interrupt that awakens two sleeping processes, one of which has higher priority than the currently executing process.

13.3 Explain what can fail if *QUANTUM* is set to 1. Hint: consider switching back to a process that was suspended by *resched* while processing an interrupt.

13.4 Does *sleepms*(3) guarantee a minimum delay of 3 milliseconds, an exact delay of 3 milliseconds, or a maximum delay of 3 milliseconds?

13.5 Explain what might happen if *wakeup* calls *wait*.

13.6 An operating system that attempts to record the exact amount of processor time a process consumes faces the following problem: when an interrupt occurs, it is most convenient to let the current process execute the interrupt routine even though the interrupt is unlikely to be related to the current process. Investigate how operating systems charge the cost of executing interrupt routines like *wakeup* to the processes that are affected.

13.7 To find out how much overhead is involved in using a high-level interrupt handler, rewrite the code to have *clkint* perform all clock interrupt handling. Devise a way to measure the extra overhead.

13.8 Consider process scheduling and code in *recvtime* carefully. Show that it is possible for *recvtime* to return a message, even if the message arrives long *after* the timer expires and the process is awakened.

13.9 Design an experiment to see if preemption ever causes the system to reschedule. Be careful: the presence of a separate process testing a variable or performing I/O can interfere with the experiment by generating calls to *resched*.

13.10 Suppose a system contains three processes: a low-priority process, L, that is sleeping, and two high-priority processes, H_1 and H_2, that are eligible to execute. Further suppose that immediately after the scheduler switches to process H_1, a clock interrupt occurs, process L becomes ready, and the interrupt handler calls *resched*. Although L will not run, *resched* will switch from H_1 to H_2 without giving H_1 its quantum. Propose a modification to *resched* that ensures a process will not lose control of the processor unless a higher-priority process becomes ready or its time slice expires.

Chapter Contents

14.1 Introduction, 259
14.2 Conceptual Organization Of I/O And Device Drivers, 260
14.3 Interface And Driver Abstractions, 261
14.4 An Example I/O Interface, 262
14.5 The Open-Read-Write-Close Paradigm, 263
14.6 Bindings For I/O Operations And Device Names, 264
14.7 Device Names In Xinu, 265
14.8 The Concept Of A Device Switch Table, 265
14.9 Multiple Copies Of A Device And Shared Drivers, 266
14.10 The Implementation Of High–level I/O Operations, 269
14.11 Other High–level I/O Functions, 271
14.12 Open, Close, And Reference Counting, 275
14.13 Null And Error Entries In Devtab, 277
14.14 Initialization Of The I/O System, 278
14.15 Perspective, 283
14.16 Summary, 283

14

Device–independent Input And Output

We have been left so much to our own devices — after a while, one welcomes the uncertainty of being left to other people's.

— Tom Stoppard

14.1 Introduction

Earlier chapters explain concurrent process support and memory management. Chapter 12 discusses the key concept of interrupts. The chapter describes interrupt processing, gives an architecture for interrupt code, and explains the relationship between interrupt handling and concurrent processes. Chapter 13 expands the discussion of interrupts by showing how real-time clock interrupts can be used to implement preemption and process delay.

This chapter takes a broader look at how an operating system implements I/O. The chapter explains the conceptual basis for building I/O abstractions, and presents an architecture for a general-purpose I/O facility. The chapter shows how processes can transfer data to or from a device without understanding the underlying hardware. It defines a general model, and explains how the model incorporates device-independent I/O functions. Finally, the chapter examines an efficient implementation of an I/O subsystem.

14.2 Conceptual Organization Of I/O And Device Drivers

Operating systems control and manage input and output (I/O) devices for three reasons. First, because most device hardware uses a low-level interface, the software interface is complex. Second, because a device is a shared resource, an operating system provides access according to policies that make sharing fair and safe. Third, an operating system defines a high-level interface that hides details and allows a programmer to use a consistent and uniform set of operations when interacting with devices.

The I/O subsystem can be divided into three conceptual pieces: an abstract interface consisting of high-level I/O functions that processes use to perform I/O, a set of physical devices, and *device driver* software that connects the two. Figure 14.1 illustrates the organization.

Figure 14.1 The conceptual organization of the I/O subsystem with device driver software between processes and the underlying device hardware.

As the figure indicates, device driver software bridges the gap between high-level, concurrent processes and low-level hardware. We will see that each driver is divided into two conceptual pieces: an *upper half* and a *lower half*. Functions in the upper half are invoked when a process requests I/O. Upper-half functions implement operations such as *read* and *write* by transferring data to or from a process. The lower half contains functions that are invoked by interrupts. When a device interrupts, the interrupt dispatcher invokes a lower-half handler function. The handler services the interrupt, interacts with the device to transfer data, and may start an additional I/O operation.

14.3 Interface And Driver Abstractions

The ultimate goal of an operating system designer lies in creating convenient programming abstractions and finding efficient implementations. With respect to I/O, there are two aspects:

- Interface abstraction
- Driver abstractions

Interface abstraction. The question arises: what I/O interface should an operating system supply to processes? There are several possibilities, and the choice represents a tradeoff among goals of flexibility, simplicity, efficiency, and generality. To understand the scope of the problem, consider Figure 14.2, which lists a set of example devices and the types of operations appropriate for each.

Device	I/O paradigm
hard drive	move to a position and transfer a block of data
keyboard	accept individual characters as entered
printer	transfer an entire document to be printed
audio output	transfer a continuous stream of encoded audio
wireless network	send or receive a single network packet

Figure 14.2 Example devices and the paradigm that each device uses.

Early operating systems provided a separate set of I/O operations for each individual hardware device. Unfortunately, building device-specific information into software means the software must be changed when an I/O device is replaced by an equivalent device from another vendor. A slightly more general approach defines a set of operations for each type of device, and requires the operating system to perform the appropriate low-level operations on a given device. For example, an operating system can offer abstract functions *send_network_packet* and *receive_network_packet* that can be used to transfer network packets over any type of network. A third approach originated in Multics and was popularized by Unix: choose a small set of abstract I/O operations that are sufficient for all I/O.

Driver abstractions. We think of the second category of abstraction as providing semantics. One of the most important semantic design questions focuses on synchrony: does a process block while waiting for an I/O operation to complete? A *synchronous* interface, similar to the one described earlier, provides blocking operations. For example, to request data from a keyboard in a synchronous system, a process invokes an

upper-half routine that blocks the process until a user presses a key. Once a user makes a keystroke, the device interrupts and the dispatcher invokes a lower-half routine that acts as a handler. The handler unblocks a waiting process and reschedules to allow the process to run. In contrast, an *asynchronous* I/O interface allows a process to continue executing after the process initiates an I/O operation. When the I/O completes, the driver must inform the requesting process (e.g., by invoking the *event handler* function associated with the process). We can summarize:

> *When using a synchronous I/O interface, a process is blocked until the operation completes. When using an asynchronous I/O interface, a process continues to execute and is notified when the operation completes.*

Each approach has advantages. An asynchronous interface is useful in situations where a programmer needs to control the overlap of I/O and computation. A synchronous approach has the advantage of being easier to program.

Another design issue arises from the format of data and the size of transfers. Two questions arise. First, will data be transferred in blocks or bytes? Second, how much data can be transferred in a single operation? Observe that some devices transfer individual data bytes, some transfer a variable-size chunk of data (such as a network packet or a line of text), and others, such as disks, transfer fixed-size blocks of data. Because a general-purpose operating system must handle a variety of I/O devices, an I/O interface may require both single-byte transfers as well as multi-byte transfers.

A final design question arises from the parameters that a driver supplies and the way a driver interprets individual operations. For example, does a process specify a location on disk and then repeatedly request the next disk block, or does the process specify a block number in each request? The example device driver presented in the next chapter illustrates the use of parameters.

The key idea is:

> *In a modern operating system, the I/O interface and device drivers are designed to hide device details and present a programmer with convenient, high-level abstractions.*

14.4 An Example I/O Interface

Experience has shown that a small set of I/O functions is both sufficient and convenient. Thus, our example system contains an I/O subsystem with nine abstract I/O operations that are used for all input and output. The operations have been derived from the I/O facilities in the Unix operating system. Figure 14.3 lists the operations and the purpose of each.

Operation	Purpose
close	Terminate use of a device
control	Perform operations other than data transfer
getc	Input a single byte of data
init	Initialize the device at system startup
open	Prepare the device for use
putc	Output a single byte of data
read	Input multiple bytes of data
seek	Move to specific data (usually a disk)
write	Output multiple bytes of data

Figure 14.3 The set of abstract I/O interface operations used in Xinu.

14.5 The Open-Read-Write-Close Paradigm

Like the programming interface in many operating systems, the example I/O interface follows an *open-read-write-close* paradigm. That is, before it can perform I/O, a process must *open* a specific device. Once it has been opened, a device is ready for the process to call *read* to obtain input or *write* to send output. Finally, once it has finished using a device, the process calls *close* to terminate use.

To summarize:

> *The open-read-write-close paradigm requires a process to open a device before use and close a device after use.*

Open and *close* allow the operating system to manage devices that require exclusive use, prepare a device for data transfer, and stop a device after transfer has ended. Closing a device may be useful, for example, if a device needs to be powered down or placed in a standby state when not in use. *Read* and *write* handle the transfer of multiple data bytes to or from a buffer in memory. *Getc* and *putc* form a counterpart for the transfer of a single byte (usually a character). *Control* allows a program to control a device or a device driver (e.g., to check supplies in a printer or select the channel on a wireless radio). *Seek* is a special case of *control* that applies to randomly accessible storage devices, such as disks. Finally, *init* initializes the device and driver at system startup.

Consider how the operations apply to a console window. *Getc* reads the next character from the keyboard, and *putc* displays one character in the console window. *Write* can display multiple characters with one call, and *read* can read a specified number of characters (or all that have been entered, depending on its arguments). Finally, *control* allows the program to change parameters in the driver to control such things as whether the system stops echoing characters as a password is entered.

14.6 Bindings For I/O Operations And Device Names

How can an abstract operation such as *read* act on an underlying hardware device? The answer lies in a binding. When a process invokes a high-level operation, the operating system maps the call to a device driver function. For example, if a process calls *read* on a console device, the operating system passes the call to a function in the console device driver that implements *read*. In doing so, the operating system hides both the hardware and device driver details from application processes and presents an abstract version of devices. By using a single abstract device for a keyboard and a window on the display, an operating system can hide the fact that the underlying hardware consists of two separate devices. Furthermore, an operating system can hide device details by presenting the same high-level abstraction for the hardware from multiple vendors. The point is:

> *An operating system creates a virtual I/O environment — a process can only perceive peripheral devices through the abstractions that the interface and device drivers provide.*

In addition to mapping abstract I/O operations onto driver routines, an operating system must map device names onto devices. A variety of mappings have been used. Early systems required a programmer to embed device names in source code. Later systems arranged for programs to use small integers to identify devices, and allowed a command interpreter to link each integer with a specific device when the application was launched. Many modern systems embed devices in a file naming hierarchy, allowing an application to use a symbolic name for each device.

Early and late binding each have advantages. An operating system that waits until runtime to bind the name of an abstract device to a real device and a set of abstract operations to device driver functions is flexible. However, such late binding systems incur more computational overhead, making them impractical in the smallest embedded systems. At the other extreme, early binding requires device information to be specified when an application is written. Thus, the essence of I/O design consists of synthesizing a binding mechanism that allows maximum flexibility within the required performance bounds.

Our example system uses an approach that is typical of small embedded systems: information about devices is specified before the operating system is compiled. For each device, the operating system knows exactly which driver functions correspond to

each of the abstract I/O operations. In addition, the operating system knows the underlying hardware device to which each abstract device corresponds. Thus, the operating system must be recompiled whenever a new device is added or when an existing device is removed. Because application code does not contain information about specific device hardware, application code can be ported from one system to another easily. For example, an application that only performs I/O operations on a *CONSOLE* serial port will work on any Xinu system that offers a *CONSOLE* device and the appropriate driver, independent of the physical device hardware and interrupt structure.

14.7 Device Names In Xinu

In Xinu, the system designer must specify a set of abstract devices when the system is configured. The configuration program assigns each device name a unique integer value known as a *device descriptor*. For example, if a designer specifies a device name *CONSOLE*, the configuration program might assign descriptor zero. The configuration program produces a header file that contains *#define* statements for each name. Thus, once the header file has been included, a programmer can reference *CONSOLE* in the code. For example, if *CONSOLE* has been assigned descriptor zero, the call:

```
read(CONSOLE, buf, 100);
```

is equivalent to:

```
read(0, buf, 100);
```

To summarize:

> *Xinu uses a static binding for device names. Each device name is bound to an integer descriptor at configuration time before the operating system is compiled.*

14.8 The Concept Of A Device Switch Table

Each time a process invokes a high-level I/O operation such as *read* or *write*, the operating system must forward the call to the appropriate driver function. To make the implementation efficient, Xinu uses an array known as a *device switch table*. The integer descriptor assigned to a device is an index into the device switch table. To understand the arrangement, imagine a two-dimensional array. Conceptually, each row of the array corresponds to a device, and each column corresponds to an abstract operation. An entry in the array specifies the driver function to use to perform the operation.

For example, suppose a system contains three devices defined as follows:

- CONSOLE, a serial device used to send and receive characters
- ETHER, an Ethernet interface device
- DISK, a hard drive

Figure 14.4 illustrates part of a device switch table that has a row for each of the three devices and a column for each I/O operation. Items in the table represent the names of driver functions that perform the operation given by the column on the device given by the row.

	open	close	read	write	getc	
CONSOLE	conopen	conclose	conread	conwrite	congetc	
ETHER	ethopen	ethclose	ethread	ethwrite	ethgetc	...
DISK	dskopen	dskclose	dskread	dskwrite	dskgetc	
			⋮			

Figure 14.4 Conceptual organization of the device switch table with one row per device and one column per abstract I/O operation.

As an example, suppose a process invokes the *write* operation on the *CONSOLE* device. The operating system goes to the row of the table that corresponds to the *CONSOLE* device, finds the column that corresponds to the *write* operation, and calls the function named in the entry, *conwrite*.

In essence, each row of the device switch table defines how the I/O operations apply to a single device, which means I/O semantics can change dramatically among devices. For example, when it is applied to a *DISK* device, a *read* might transfer a block of 512 bytes of data. However, when it is applied to a *CONSOLE* device, *read* might transfer a line of characters that the user has entered.

The most significant aspect of the device switch table arises from the ability to define a uniform abstraction across multiple physical devices. For example, suppose a computer contains a disk that uses 1 Kbyte sectors and a disk that uses 4 Kbyte sectors. Drivers for the two disks can present an identical interface to applications, while hiding the differences in the underlying hardware. That is, a driver can always transfer 4 Kbytes to a user, and convert each transfer into four 1 Kbyte disk transfers.

14.9 Multiple Copies Of A Device And Shared Drivers

Suppose a given computer has two devices that use identical hardware. Does the operating system need two separate copies of the device driver? No. The system contains one copy of each driver routine and uses a parameter to distinguish between the two devices. Parameters are kept in columns of the device switch table in addition to the functions that Figure 14.4 illustrates. For example, if a system contains two Ethernet interfaces, each will have its own row in the device switch table. Most entries in the two rows will be identical. However, one column will specify a unique *Control and Status Register* (*CSR*) address for each device. When it invokes a driver function, the

system passes an argument that contains a pointer to the row in the device switch table for the device. Thus, a driver function can apply the operation to the correct device. The point is:

> *Instead of creating a device driver for each physical device, an operating system maintains a single copy of the driver for each type of device and supplies an argument that permits the driver to distinguish among multiple copies of the physical hardware.*

A look at the definition of the device switch table, *devtab*, will clarify the details. Structure *dentry* defines the format of entries in the table; the declaration can be found in file *conf.h*†.

```
/* conf.h (GENERATED FILE; DO NOT EDIT) */

/* Device switch table declarations */

/* Device table entry */
struct  dentry  {
        int32   dvnum;
        int32   dvminor;
        char    *dvname;
        devcall (*dvinit) (struct dentry *);
        devcall (*dvopen) (struct dentry *, char *, char *);
        devcall (*dvclose)(struct dentry *);
        devcall (*dvread) (struct dentry *, void *, uint32);
        devcall (*dvwrite)(struct dentry *, void *, uint32);
        devcall (*dvseek) (struct dentry *, int32);
        devcall (*dvgetc) (struct dentry *);
        devcall (*dvputc) (struct dentry *, char);
        devcall (*dvcntl) (struct dentry *, int32, int32, int32);
        void    *dvcsr;
        void    (*dvintr)(void);
        byte    dvirq;
};

extern  struct  dentry  devtab[]; /* one entry per device */

/* Device name definitions */

#define CONSOLE     0       /* type tty      */
#define NULLDEV     1       /* type null     */
#define ETHER0      2       /* type eth      */
```

†Chapter 25 explains Xinu configuration and gives more detail about *conf.h*.

```
#define NAMESPACE   3       /* type nam      */
#define RDISK       4       /* type rds      */
#define RAM0        5       /* type ram      */
#define RFILESYS    6       /* type rfs      */
#define RFILE0      7       /* type rfl      */
#define RFILE1      8       /* type rfl      */
#define RFILE2      9       /* type rfl      */
#define RFILE3      10       /* type rfl      */
#define RFILE4      11       /* type rfl      */
#define RFILE5      12       /* type rfl      */
#define RFILE6      13       /* type rfl      */
#define RFILE7      14       /* type rfl      */
#define RFILE8      15       /* type rfl      */
#define RFILE9      16       /* type rfl      */
#define LFILESYS    17       /* type lfs      */
#define LFILE0      18       /* type lfl      */
#define LFILE1      19       /* type lfl      */
#define LFILE2      20       /* type lfl      */
#define LFILE3      21       /* type lfl      */
#define LFILE4      22       /* type lfl      */
#define LFILE5      23       /* type lfl      */

/* Control block sizes */

#define Nnull   1
#define Ntty    1
#define Neth    1
#define Nrds    1
#define Nram    1
#define Nrfs    1
#define Nrfl    10
#define Nlfs    1
#define Nlfl    6
#define Nnam    1

#define DEVMAXNAME 24
#define NDEVS 24


/* Configuration and Size Constants */

#define NPROC        100        /* number of user processes            */
#define NSEM         100        /* number of semaphores                */
#define IRQBASE      32         /* base ivec for IRQ0                  */
#define IRQ_TIMER    IRQ_HW5    /* timer IRQ is wired to hardware 5    */
#define IRQ_ATH_MISC IRQ_HW4    /* Misc. IRQ is wired to hardware 4    */
#define CLKFREQ      200000000  /* 200 MHz clock                       */
```

```
#define LF_DISK_DEV     RAM0
```

Each entry in *devtab* corresponds to a single device. The entry specifies the address of functions that constitute the driver for the device, the device CSR address, and other information used by the driver. Fields *dvinit*, *dvopen*, *dvclose*, *dvread*, *dvwrite*, *dvseek*, *dvgetc*, *dvputc*, and *dvcntl* hold the addresses of driver routines that correspond to high-level operations. Field *dvminor* contains an integer index into the control block array for the device. A minor device number accommodates multiple identical hardware devices by allowing a driver to maintain a separate control block entry for each physical device. Field *dvcsr* contains the hardware CSR address for the device. The control block for a device holds additional information for the particular instance of the device and the driver; the contents depend on the device, but may include such things as input or output buffers, device status information (e.g., whether a wireless networking device is currently in contact with another wireless device), and accounting information (e.g., the total amount of data sent or received since the system booted).

14.10 The Implementation Of High–level I/O Operations

Because it isolates high-level I/O operations from underlying details, the device switch table allows high-level functions to be created before any device drivers have been written. One of the chief benefits of such a strategy arises because a programmer can build pieces of the I/O system without requiring specific hardware devices to be present.

The example system contains a function for each high-level I/O operation. Thus, the system contains functions *open*, *close*, *read*, *write*, *getc*, *putc*, and so on. However, a function such as *read* does not perform I/O. Instead, each high-level I/O function operates *indirectly*: the function uses the device switch table to find and invoke the appropriate low-level device driver routine to perform the requested function. The point is:

> *Instead of performing I/O, high-level functions such as* read *and* write *use a level of indirection to invoke a low-level driver function for the specified device*.

An examination of the code will clarify the concept. Consider the *read* function found in file *read.c*:

```
/* read.c - read */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  read  -  Read one or more bytes from a device
 *------------------------------------------------------------------------
 */
syscall read(
          did32         descrp,         /* Descriptor for device        */
          char          *buffer,        /* Address of buffer            */
          uint32        count           /* Length of buffer             */
        )
{
        intmask         mask;           /* Saved interrupt mask         */
        struct dentry   *devptr;        /* Entry in device switch table */
        int32           retval;         /* Value to return to caller    */

        mask = disable();
        if (isbaddev(descrp)) {
                restore(mask);
                return SYSERR;
        }
        devptr = (struct dentry *) &devtab[descrp];
        retval = (*devptr->dvread) (devptr, buffer, count);
        restore(mask);
        return retval;
}
```

The arguments to *read* consist of a *device descriptor*, the address of a buffer, and an integer that gives the maximum number of bytes to read. *Read* uses the device descriptor, *descrp*, as an index into *devtab*, and assigns pointer *devptr* the address of the device switch table entry. The *return* statement contains code that performs the task of invoking the underlying device driver function and returning the result to the function that called *read*. The code:

(*devptr->dvread) (devptr, buffer, count)

performs the indirect function call. That is, the code invokes the driver function given by field *dvread* in the device switch table entry, passing the function three arguments: the address of the *devtab* entry, *devptr*, the buffer address, *buffer*, and a count of characters to read, *count*.

14.11 Other High–level I/O Functions

The remaining high-level transfer and control functions operate exactly as *read*: they use the device switch table to select the appropriate low-level driver function, invoke the function, and return the result to the caller. Code for each function is shown below.

```
/* control.c - control */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  control  -  Control a device or a driver (e.g., set the driver mode)
 *------------------------------------------------------------------------
 */
syscall control(
          did32         descrp,         /* Descriptor for device        */
          int32         func,           /* Specific control function    */
          int32         arg1,           /* Specific argument for func   */
          int32         arg2            /* Specific argument for func   */
        )
{
        intmask         mask;           /* Saved interrupt mask         */
        struct dentry   *devptr;        /* Entry in device switch table */
        int32           retval;         /* Value to return to caller    */

        mask = disable();
        if (isbaddev(descrp)) {
                restore(mask);
                return SYSERR;
        }
        devptr = (struct dentry *) &devtab[descrp];
        retval = (*devptr->dvcntl) (devptr, func, arg1, arg2);
        restore(mask);
        return retval;
}
```

```
/* getc.c - getc */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  getc  -  Obtain one byte from a device
 *------------------------------------------------------------------------
 */
syscall getc(
          did32         descrp          /* Descriptor for device        */
        )
{
        intmask         mask;           /* Saved interrupt mask         */
        struct dentry   *devptr;        /* Entry in device switch table */
        int32           retval;         /* Value to return to caller    */

        mask = disable();
        if (isbaddev(descrp)) {
                restore(mask);
                return SYSERR;
        }
        devptr = (struct dentry *) &devtab[descrp];
        retval = (*devptr->dvgetc) (devptr);
        restore(mask);
        return retval;
}

/* putc.c - putc */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  putc  -  Send one character of data (byte) to a device
 *------------------------------------------------------------------------
 */
syscall putc(
          did32         descrp,         /* Descriptor for device        */
          char          ch              /* Character to send            */
        )
{
        intmask         mask;           /* Saved interrupt mask         */
        struct dentry   *devptr;        /* Entry in device switch table */
        int32           retval;         /* Value to return to caller    */
```

```
        mask = disable();
        if (isbaddev(descrp)) {
                restore(mask);
                return SYSERR;
        }
        devptr = (struct dentry *) &devtab[descrp];
        retval = (*devptr->dvputc) (devptr, ch);
        restore(mask);
        return retval;
}

/* seek.c - seek */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  seek  -  Position a random access device
 *------------------------------------------------------------------------
 */
syscall seek(
          did32         descrp,         /* Descriptor for device        */
          uint32        pos             /* Position                     */
        )
{
        intmask         mask;           /* Saved interrupt mask         */
        struct dentry   *devptr;        /* Entry in device switch table */
        int32           retval;         /* Value to return to caller    */

        mask = disable();
        if (isbaddev(descrp)) {
                restore(mask);
                return SYSERR;
        }
        devptr = (struct dentry *) &devtab[descrp];
        retval = (*devptr->dvseek) (devptr, pos);
        restore(mask);
        return retval;
}
```

```
/* write.c - write */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  write  -  Write one or more bytes to a device
 *------------------------------------------------------------------------
 */
syscall write(
          did32         descrp,         /* Descriptor for device        */
          char          *buffer,        /* Address of buffer            */
          uint32        count           /* Length of buffer             */
        )
{
        intmask         mask;           /* Saved interrupt mask         */
        struct dentry   *devptr;        /* Entry in device switch table */
        int32           retval;         /* Value to return to caller    */

        mask = disable();
        if (isbaddev(descrp)) {
                restore(mask);
                return SYSERR;
        }
        devptr = (struct dentry *) &devtab[descrp];
        retval = (*devptr->dvwrite) (devptr, buffer, count);
        restore(mask);
        return retval;
}
```

The functions listed above are designed to allow user processes to access I/O devices. In addition, the system provides one high-level I/O function that is intended only for the operating system to use: *init*. We will see that when it boots, the operating system calls *init* for each device. Like the other I/O functions, *init* uses the device switch table to invoke the appropriate low-level driver function. Thus, the initialization function in each driver can initialize the hardware device, if necessary, and can also initialize the data structures used by the driver (e.g., buffers and semaphores). We will see examples of driver initialization later. For now, it is sufficient to understand that *init* follows the same approach as other I/O functions:

```
/* init.c - init */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  init  -  Initialize a device and its driver
 *------------------------------------------------------------------------
 */
syscall init(
          did32         descrp          /* Descriptor for device        */
        )
{
        intmask         mask;           /* Saved interrupt mask         */
        struct dentry   *devptr;        /* Entry in device switch table */
        int32           retval;         /* Value to return to caller    */

        mask = disable();
        if (isbaddev(descrp)) {
                restore(mask);
                return SYSERR;
        }
        devptr = (struct dentry *) &devtab[descrp];
        retval = (*devptr->dvinit) (devptr);
        restore(mask);
        return retval;
}
```

14.12 Open, Close, And Reference Counting

Functions *open* and *close* operate similar to other I/O functions by using the device switch table to call the appropriate driver function. One motivation for using *open* and *close* arises from their ability to establish ownership of a device or prepare a device for use. For example, if a device requires exclusive access, *open* can block a subsequent user until the device becomes free. As another example, consider a system that saves power by keeping a disk device idle when the device is not in use. Although a designer could arrange to use the *control* function to start or stop a disk, *open* and *close* are more convenient. Thus, a disk can be powered on when a process calls *open*, and powered off when a process calls *close*.

Although a small embedded system might choose to power down a disk whenever a process calls *close* on the device, larger systems need a more sophisticated mechanism because multiple processes can use a device simultaneously. Thus, most drivers employ a technique known as *reference counting*. That is, a driver maintains an integer variable

that counts the number of processes using the device. During initialization, the reference count is set to zero. Whenever a process calls *open*, the driver increments the reference count, and whenever a process calls *close*, the driver decrements the reference count. When the reference count reaches zero, the driver powers down the device.

The code for *open* and *close* follows the same approach as the code for other high-level I/O functions:

```
/* open.c - open */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  open  -  Open a device (some devices ignore name and mode parameters)
 *------------------------------------------------------------------------
 */
syscall open(
          did32         descrp,         /* Descriptor for device        */
          char          *name,          /* Name to use, if any          */
          char          *mode           /* Mode for device, if any      */
        )
{
        intmask         mask;           /* Saved interrupt mask         */
        struct dentry   *devptr;        /* Entry in device switch table */
        int32           retval;         /* Value to return to caller    */

        mask = disable();
        if (isbaddev(descrp)) {
                restore(mask);
                return SYSERR;
        }
        devptr = (struct dentry *) &devtab[descrp];
        retval = (*devptr->dvopen) (devptr, name, mode);
        restore(mask);
        return retval;
}

/* close.c - close */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  close  -  Close a device
 *------------------------------------------------------------------------
 */
```

```
syscall close(
          did32         descrp          /* Descriptor for device        */
        )
{
        intmask         mask;           /* Saved interrupt mask         */
        struct  dentry  *devptr;        /* Entry in device switch table */
        int32           retval;         /* Value to return to caller    */

        mask = disable();
        if (isbaddev(descrp)) {
                restore(mask);
                return SYSERR;
        }
        devptr = (struct dentry *) &devtab[descrp];
        retval = (*devptr->dvclose) (devptr);
        restore(mask);
        return retval;
}
```

14.13 Null And Error Entries In Devtab

An interesting dilemma arises from the way I/O functions operate. On the one hand, high-level functions, such as *read* and *write*, use entries in *devtab* without checking whether the entries are valid. Thus, a function must be supplied for every I/O operation for each device. On the other hand, an operation may not be meaningful on all devices. For example, *seek* is not an operation that can be performed on a serial device, and *getc* is not meaningful on a network device that delivers packets. Furthermore, a designer may choose to ignore an operation on a particular device (e.g., a designer may choose to leave the CONSOLE device open at all times, which means the *close* operation has no effect).

What value can be used in *devtab* for operations that are not meaningful? The answer lies in two special routines that can be used to fill in entries of *devtab* that have no driver functions:

- ionull — return OK without performing any action
- ioerr — return SYSERR without performing any action

By convention, entries filled with *ioerr* should never be called; they signify an illegal operation. Entries for unnecessary, but otherwise innocuous operations (e.g., *open* for a terminal device), point to function *ionull*. The code for each of the two functions is trivial.

```
/* ionull.c - ionull */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ionull  -  Do nothing (used for "don't care" entries in devtab)
 *------------------------------------------------------------------------
 */
devcall ionull(void)
{
        return OK;
}

/* ioerr.c - ioerr */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ioerr  -  Return an error status (used for "error" entries in devtab)
 *------------------------------------------------------------------------
 */
devcall ioerr(void)
{
        return SYSERR;
}
```

14.14 Initialization Of The I/O System

How is a device switch table initialized? How are driver functions installed? In a large, complex operating system, device drivers can be managed dynamically. Thus, when it finds new device hardware, the operating system can identify the hardware, search for an appropriate driver, and install the driver without rebooting.

A small embedded system does not have a collection of drivers available on secondary storage, and may not have sufficient computational resources to install drivers at runtime. Thus, most embedded systems use a static device configuration in which the set of devices and the set of device drivers are specified when the system is compiled. Our example follows the static approach by requiring the system designer to specify a set of devices and the set of low-level driver functions that constitute each driver. Instead of forcing a programmer to enter explicit declarations for the entire device switch table, however, a separate application program is used that reads a configuration file and generates a C file that contains a declaration of *devtab* with an initial value for each field. The point is:

Small embedded systems use a static device specification in which a designer specifies the set of devices plus the device driver functions for each; a configuration program can generate code that assigns a value to each field of the device switch table.

File *conf.c* contains an example of the C code generated by the configuration program. Chapter 25 explains how *conf.c* is generated; for now, it is sufficient to examine one of the entries in *devtab* and observe how each field is initialized.

```
/* conf.c (GENERATED FILE; DO NOT EDIT) */

#include <xinu.h>


extern  devcall ioerr(void);
extern  devcall ionull(void);

/* Device independent I/O switch */

struct  dentry  devtab[NDEVS] =
{
/**
 * Format of entries is:
 * dev-number, minor-number, dev-name,
 * init, open, close,
 * read, write, seek,
 * getc, putc, control,
 * dev-csr-address, intr-handler, irq
 */

/* CONSOLE is tty */
        { 0, 0, "CONSOLE",
          (void *)ttyinit, (void *)ionull, (void *)ionull,
          (void *)ttyread, (void *)ttywrite, (void *)ioerr,
          (void *)ttygetc, (void *)ttyputc, (void *)ttycontrol,
          (void *)0x3f8, (void *)ttydispatch, 42 },

/* NULLDEV is null */
        { 1, 0, "NULLDEV",
          (void *)ionull, (void *)ionull, (void *)ionull,
          (void *)ionull, (void *)ionull, (void *)ioerr,
          (void *)ionull, (void *)ionull, (void *)ioerr,
          (void *)0x0, (void *)ioerr, 0 },
```

```
/* ETHER0 is eth */
        { 2, 0, "ETHER0",
          (void *)ethinit, (void *)ioerr, (void *)ioerr,
          (void *)ethread, (void *)ethwrite, (void *)ioerr,
          (void *)ioerr, (void *)ioerr, (void *)ethcontrol,
          (void *)0x0, (void *)ethdispatch, 43 },

/* NAMESPACE is nam */
        { 3, 0, "NAMESPACE",
          (void *)naminit, (void *)namopen, (void *)ioerr,
          (void *)ioerr, (void *)ioerr, (void *)ioerr,
          (void *)ioerr, (void *)ioerr, (void *)ioerr,
          (void *)0x0, (void *)ioerr, 0 },

/* RDISK is rds */
        { 4, 0, "RDISK",
          (void *)rdsinit, (void *)rdsopen, (void *)rdsclose,
          (void *)rdsread, (void *)rdswrite, (void *)ioerr,
          (void *)ioerr, (void *)ioerr, (void *)rdscontrol,
          (void *)0x0, (void *)ionull, 0 },

/* RAM0 is ram */
        { 5, 0, "RAM0",
          (void *)raminit, (void *)ramopen, (void *)ramclose,
          (void *)ramread, (void *)ramwrite, (void *)ioerr,
          (void *)ioerr, (void *)ioerr, (void *)ioerr,
          (void *)0x0, (void *)ionull, 0 },

/* RFILESYS is rfs */
        { 6, 0, "RFILESYS",
          (void *)rfsinit, (void *)rfsopen, (void *)ioerr,
          (void *)ioerr, (void *)ioerr, (void *)ioerr,
          (void *)ioerr, (void *)ioerr, (void *)rfscontrol,
          (void *)0x0, (void *)ionull, 0 },

/* RFILE0 is rfl */
        { 7, 0, "RFILE0",
          (void *)rflinit, (void *)ioerr, (void *)rflclose,
          (void *)rflread, (void *)rflwrite, (void *)rflseek,
          (void *)rflgetc, (void *)rflputc, (void *)ioerr,
          (void *)0x0, (void *)ionull, 0 },

/* RFILE1 is rfl */
        { 8, 1, "RFILE1",
          (void *)rflinit, (void *)ioerr, (void *)rflclose,
```

```
          (void *)rflread, (void *)rflwrite, (void *)rflseek,
          (void *)rflgetc, (void *)rflputc, (void *)ioerr,
          (void *)0x0, (void *)ionull, 0 },

/* RFILE2 is rfl */
        { 9, 2, "RFILE2",
          (void *)rflinit, (void *)ioerr, (void *)rflclose,
          (void *)rflread, (void *)rflwrite, (void *)rflseek,
          (void *)rflgetc, (void *)rflputc, (void *)ioerr,
          (void *)0x0, (void *)ionull, 0 },

/* RFILE3 is rfl */
        { 10, 3, "RFILE3",
          (void *)rflinit, (void *)ioerr, (void *)rflclose,
          (void *)rflread, (void *)rflwrite, (void *)rflseek,
          (void *)rflgetc, (void *)rflputc, (void *)ioerr,
          (void *)0x0, (void *)ionull, 0 },

/* RFILE4 is rfl */
        { 11, 4, "RFILE4",
          (void *)rflinit, (void *)ioerr, (void *)rflclose,
          (void *)rflread, (void *)rflwrite, (void *)rflseek,
          (void *)rflgetc, (void *)rflputc, (void *)ioerr,
          (void *)0x0, (void *)ionull, 0 },

/* RFILE5 is rfl */
        { 12, 5, "RFILE5",
          (void *)rflinit, (void *)ioerr, (void *)rflclose,
          (void *)rflread, (void *)rflwrite, (void *)rflseek,
          (void *)rflgetc, (void *)rflputc, (void *)ioerr,
          (void *)0x0, (void *)ionull, 0 },

/* RFILE6 is rfl */
        { 13, 6, "RFILE6",
          (void *)rflinit, (void *)ioerr, (void *)rflclose,
          (void *)rflread, (void *)rflwrite, (void *)rflseek,
          (void *)rflgetc, (void *)rflputc, (void *)ioerr,
          (void *)0x0, (void *)ionull, 0 },

/* RFILE7 is rfl */
        { 14, 7, "RFILE7",
          (void *)rflinit, (void *)ioerr, (void *)rflclose,
          (void *)rflread, (void *)rflwrite, (void *)rflseek,
          (void *)rflgetc, (void *)rflputc, (void *)ioerr,
          (void *)0x0, (void *)ionull, 0 },
```

```
/* RFILE8 is rfl */
        { 15, 8, "RFILE8",
          (void *)rflinit, (void *)ioerr, (void *)rflclose,
          (void *)rflread, (void *)rflwrite, (void *)rflseek,
          (void *)rflgetc, (void *)rflputc, (void *)ioerr,
          (void *)0x0, (void *)ionull, 0 },

/* RFILE9 is rfl */
        { 16, 9, "RFILE9",
          (void *)rflinit, (void *)ioerr, (void *)rflclose,
          (void *)rflread, (void *)rflwrite, (void *)rflseek,
          (void *)rflgetc, (void *)rflputc, (void *)ioerr,
          (void *)0x0, (void *)ionull, 0 },

/* LFILESYS is lfs */
        { 17, 0, "LFILESYS",
          (void *)lfsinit, (void *)lfsopen, (void *)ioerr,
          (void *)ioerr, (void *)ioerr, (void *)ioerr,
          (void *)ioerr, (void *)ioerr, (void *)ioerr,
          (void *)0x0, (void *)ionull, 0 },

/* LFILE0 is lfl */
        { 18, 0, "LFILE0",
          (void *)lflinit, (void *)ioerr, (void *)lflclose,
          (void *)lflread, (void *)lflwrite, (void *)lflseek,
          (void *)lflgetc, (void *)lflputc, (void *)lflcontrol,
          (void *)0x0, (void *)ionull, 0 },

/* LFILE1 is lfl */
        { 19, 1, "LFILE1",
          (void *)lflinit, (void *)ioerr, (void *)lflclose,
          (void *)lflread, (void *)lflwrite, (void *)lflseek,
          (void *)lflgetc, (void *)lflputc, (void *)lflcontrol,
          (void *)0x0, (void *)ionull, 0 },

/* LFILE2 is lfl */
        { 20, 2, "LFILE2",
          (void *)lflinit, (void *)ioerr, (void *)lflclose,
          (void *)lflread, (void *)lflwrite, (void *)lflseek,
          (void *)lflgetc, (void *)lflputc, (void *)lflcontrol,
          (void *)0x0, (void *)ionull, 0 },

/* LFILE3 is lfl */
        { 21, 3, "LFILE3",
```

```
          (void *)lflinit, (void *)ioerr, (void *)lflclose,
          (void *)lflread, (void *)lflwrite, (void *)lflseek,
          (void *)lflgetc, (void *)lflputc, (void *)lflcontrol,
          (void *)0x0, (void *)ionull, 0 },

/* LFILE4 is lfl */
        { 22, 4, "LFILE4",
          (void *)lflinit, (void *)ioerr, (void *)lflclose,
          (void *)lflread, (void *)lflwrite, (void *)lflseek,
          (void *)lflgetc, (void *)lflputc, (void *)lflcontrol,
          (void *)0x0, (void *)ionull, 0 },

/* LFILE5 is lfl */
        { 23, 5, "LFILE5",
          (void *)lflinit, (void *)ioerr, (void *)lflclose,
          (void *)lflread, (void *)lflwrite, (void *)lflseek,
          (void *)lflgetc, (void *)lflputc, (void *)lflcontrol,
          (void *)0x0, (void *)ionull, 0 }
};
```

14.15 Perspective

Device-independent I/O is now an integral part of mainstream computing, and the advantages seem obvious. However, it took decades for the computing community to reach consensus on device-independent I/O and to devise a set of primitives. Some of the contention arose because programming languages each define a set of I/O abstractions. For example, FORTRAN used device numbers and required a mechanism that could bind each number to an I/O device or file. Operating system designers wanted to accommodate all languages because a large volume of code has been written in each. So, questions arise. Have we chosen the best set of device-independent I/O functions, or have we merely become so accustomed to using them that we fail to look for alternatives? Are the functions we are using ideal for systems that focus on graphical devices?

14.16 Summary

An operating system hides the details of peripheral devices, and provides a set of abstract, device-independent functions that can be used to perform I/O. The example system uses nine abstract functions: *open*, *close*, *control*, *getc*, *putc*, *read*, *write*, *seek*, and an initialization function, *init*. In our design, each of the I/O primitives operates *synchronously*, delaying a calling process until the request has been satisfied (e.g., function *read* delays the calling process until data has arrived).

The example system defines an abstract device name (such as *CONSOLE*) for each device, and assigns the device a unique integer device descriptor. The system uses a device switch table to bind a descriptor to a specific device at runtime. Conceptually, the device switch table contains one row for each device and one column for each

abstract I/O operation; additional columns point to a control block for the device, and a minor device number is used to distinguish among multiple copies of a physical device. A high-level I/O operation, such as *read* or *write*, uses the device switch table to invoke the device driver function that performs the requested operation on the specified device. Individual drivers interpret the calls in a way meaningful to a particular device; if an operation makes no sense when applied to a particular device, the device switch table is configured to invoke function *ioerr*, which returns an error code.

EXERCISES

14.1 Identify the set of abstract I/O operations available in Linux.

14.2 Find a system that uses *asynchronous* I/O, and identify the mechanism by which a running program is notified when the operation completes. Which approach, synchronous or asynchronous, makes it easier to program? Explain.

14.3 The chapter discusses two separate bindings: the binding from a device name (e.g., *CONSOLE*) to a descriptor (e.g., 0) and the binding from a device descriptor to a specific hardware device. Explain how Linux performs the two bindings.

14.4 Consider the implementation of device names in the example code. Is it possible to write a program that allows a user to enter a device name (e.g., *CONSOLE*), and then open the device? Why or why not?

14.5 Assume that in the course of debugging you begin to suspect that a process is making incorrect calls to high-level I/O functions (e.g., calling *seek* on a device for which the operation makes no sense). How can you make a quick change to the code to intercept such errors and display the process ID of the offending process? (Make the change without recompiling the source code.)

14.6 Are the abstract I/O operations presented in the chapter sufficient for all I/O operations? Explain. Hint: consider socket functions found in Unix.

14.7 Xinu defines the device subsystem as the fundamental I/O abstraction and merges files into the device system. Unix systems define the file system as the fundamental abstraction and merge devices into the file system. Compare the two approaches and list the advantages of each.

Chapter Contents

15.1 Introduction, 287
15.2 Serial Communication Using UART Hardware, 287
15.3 The Tty Abstraction, 288
15.4 Organization Of A Tty Device Driver, 289
15.5 Request Queues And Buffers, 290
15.6 Synchronization Of Upper Half And Lower Half, 291
15.7 UART Hardware FIFOs And Driver Design, 292
15.8 The Concept Of A Control Block, 293
15.9 Tty Control Block And Data Declarations, 293
15.10 Minor Device Numbers, 296
15.11 Upper–half Tty Character Input (ttygetc), 297
15.12 Upper–half Tty Read Function (ttyread), 298
15.13 Upper–half Tty Character Output (ttyputc), 300
15.14 Starting Output (ttykickout), 301
15.15 Upper–half Tty Multiple Character Output (ttywrite), 302
15.16 Lower–half Tty Driver Function (ttyhandler), 303
15.17 Output Interrupt Processing (ttyhandle_out), 306
15.18 Tty Input Processing (ttyhandle_in), 308
15.19 Tty Control Block Initialization (ttyinit), 315
15.20 Device Driver Control (ttycontrol), 317
15.21 Perspective, 319
15.22 Summary, 320

15

An Example Device Driver

It's hard to find a good driver these days, one with character and style.

— Unknown

15.1 Introduction

Chapters in this section of the text explore the general structure of an I/O system, including interrupt processing and real-time clock management. The previous chapter presents the organization of the I/O subsystem, a set of abstract I/O operations, and an efficient implementation using a device switch table.

This chapter continues the exploration of I/O. The chapter explains how a driver can define an I/O service at a high level of abstraction that is independent of the underlying hardware. The chapter also elaborates on the conceptual division of a device driver into upper and lower halves by explaining how the two halves share data structures, such as buffers, and how they communicate. Finally, the chapter shows the details of a particular example: a driver for an asynchronous character-oriented serial device.

15.2 Serial Communication Using UART Hardware

The console line on most embedded systems uses a *Universal Asynchronous Transmitter and Receiver (UART*) chip that implements serial communication according to the RS-232 standard. UART hardware is primitive — it only provides the ability to send and receive individual bytes. It does not interpret the meaning of bytes or provide functions such as the use of backspace to erase previous input.

15.3 The Tty Abstraction

Xinu uses the name *tty* to refer to the abstraction of an interface used with a character-oriented text window that displays characters the user enters on a serial interface, such as a keyboard.† In broad terms, a tty device supports two-way communication: a process can send characters to the output side and/or receive characters from the input side. Although the underlying serial hardware mechanism operates the input and output independently, the tty abstraction allows the two to be treated as a single mechanism in which the input and output appear to be connected. For example, our tty driver supports *character echo*, which means that the input side of the driver can output a copy of each incoming character. Echo is especially important when a user is typing on a keyboard and expects to see characters displayed on a screen.

The tty abstraction illustrates an important feature of many device drivers: multiple *modes*. At any time, the driver operates in one mode, and the mode can be changed at runtime. Figure 15.1 summarizes the three modes our driver supports.

Mode	Meaning
raw	The driver delivers each incoming character as it arrives without echoing the character, buffering a line of text, performing translation, or controlling the output flow
cooked	The driver buffers input, echoes characters in a readable form, honors backspace and line kill, allows type-ahead, handles flow control, and delivers an entire line of text
cbreak	The driver handles character translation, echoing, and flow control, but instead of buffering an entire line of text, the driver delivers each incoming character as it arrives

Figure 15.1 Three modes supported by the tty abstraction.

Raw mode is intended to give applications access to input characters with no preprocessing. In raw mode, the tty driver merely delivers input without interpreting or changing characters. The driver does not echo characters nor does it handle flow control. Raw mode is useful when handling non-interactive communication, such as downloading a binary file over a serial line or using a serial device to control a sensor.

Cooked mode handles interactive keyboard input. Each time it receives a character, the driver echoes the character (i.e., transmits a copy of the character to the output), which allows a user to see characters as they are typed. The driver has a parameter to control character echoing, which means an application can turn echo off and back on (e.g., to prompt for a password). Cooked mode supports *line buffering*, which means that the driver collects all characters of a line before delivering them to a reading proc-

†The name *tty* is taken from early Unix systems that used an ASCII Teletype device that consisted of a keyboard and an associated printer mechanism.

ess. Because the tty driver performs character echo and other functions at interrupt time, a user can type ahead, even if no application is reading characters (e.g., a user can type the next command while the current command is running). A chief advantage of cooked mode arises from the ability to edit a line, either by backspacing or by typing a special character to erase the entire line.

Cooked mode also provides *flow control* and *input mapping*. When enabled, flow control allows a user to type *control-s* to stop output temporarily, and later to type *control-q* to restart output. Input mapping handles hardware (or applications) that use the two-character sequence of *carriage return* (*cr*) and *linefeed* (*lf*) to terminate a line of text instead of a single *lf*. Cooked mode contains a *crlf*† parameter that controls how the driver handles line termination. When a user presses the key labeled *ENTER* or *RETURN*, the driver consults the parameter to decide whether to pass a *linefeed* (also called a *NEWLINE*) character to the application or to map the *linefeed* into a pair of characters, *carriage return* followed by *linefeed*.

Cbreak mode provides a compromise between cooked and raw modes. In cbreak mode, each character is delivered to the application instantly, without waiting to accumulate a line of text. Thus, the driver does not buffer input, nor does the driver support backspace or line kill functions. However, the driver does handle both character echo and flow control.

15.4 Organization Of A Tty Device Driver

Like most device drivers, the example tty driver is partitioned into an *upper half* that contains functions called by application processes (indirectly through the device switch table), and a *lower half* that contains functions invoked when the device interrupts. The two halves share a data structure that contains information about the device, the current mode of the driver, and buffers for incoming and outgoing data. In general, upper-half functions move data to or from the shared structure and have minimal interaction with the device hardware. For example, an upper-half function places outgoing data in the shared structure where a lower-half function can access and send the data to the device. Similarly, the lower half places incoming data in the shared structure where an upper-half function can extract it.

The motivation for driver partitioning can be difficult to appreciate at first. We will see, however, that dividing a driver into two halves is fundamental because the division allows a system designer to decouple normal processing from hardware interrupt processing and understand exactly how each function is invoked. The point is:

> *When creating a device driver, a programmer must be careful to preserve the division between upper-half and lower-half functions because upper-half functions are called by application processes and lower-half functions are invoked by interrupts.*

†Pronounced *curl-if*.

15.5 Request Queues And Buffers

In most drivers, the shared data structure contains two key items:

- Request queue
- Buffered I/O

Request queue. In principle, the most important item found in the data structure shared by upper-half and lower-half functions is a queue into which the upper half places requests. Conceptually, the *request queue* connects high-level operations that applications specify and low-level actions that must be performed on the device. Each driver has its own set of requests, and the contents of elements on a request queue depend on the underlying device as well as the operations to be performed. For example, requests issued to a disk device specify the direction of transfer (read or write), a location on the disk, and a buffer for the data to be transferred. Requests issued to a network device might specify a set of packet buffers, and specify whether to transmit a packet from the buffer or receive an incoming packet. Our example driver has two queues: one contains a set of outgoing characters and the other holds incoming characters. In essence, a character in an outgoing queue is a request to transmit, and space in the input queue serves as a request to receive.

Buffered I/O. Most drivers buffer incoming and outgoing data. Output buffering allows an application to deposit an outgoing item in a buffer, and then continue processing. The item remains in the buffer until the hardware is ready to accept the item. Input buffering allows a driver to accept data from a device before an application is ready to receive it. An incoming item remains in the buffer from the time the device deposits the item until a process requests it.

Buffers are important for several reasons. First, a driver can accept incoming data before a user process consumes the data. Input buffering is especially important for asynchronous devices like a network interface that can receive packets at any time or a keyboard on which a user can press a key at any time. Second, buffering permits an application to read or write arbitrary amounts of data, even if the underlying hardware transfers entire blocks. By placing a block in a buffer, an operating system can satisfy subsequent requests from the buffer without an I/O transfer. Third, output buffering permits the driver to perform I/O concurrently with processing because a process can write data into a buffer, and then continue executing while the driver writes the data to the device.

Our tty driver uses circular input and output buffers, with an extra circular buffer for echoed characters (echoed characters are kept in a buffer separate from normal output because echoed characters have higher priority). We think of each buffer as a conceptual queue, with characters being inserted at the tail and removed from the head. Figure 15.2 illustrates the concept of a circular output buffer, and shows the implementation with an array of bytes in memory.

Figure 15.2 (a) A circular output buffer acting as a queue, and (b) the implementation with an array of bytes.

Output functions deposit characters to be sent in the output buffer and return to their caller. When it places characters in the output buffer, an upper-half function must also start output interrupts on the device. Whenever the device generates an output interrupt, the lower half extracts up to sixteen characters from the output buffer, and deposits the characters in the device's output FIFO.† Once all characters in the output FIFO have been transmitted, the device will interrupt again. Thus, output continues until the output buffer becomes empty at which time the driver stops output and the device becomes idle.

Input works the other way around. Whenever it receives characters, the device interrupts and the interrupt dispatcher calls a lower-half function (i.e., *ttyhandler*). The interrupt handler extracts the characters from the device's input FIFO and deposits them in the circular input buffer. When a process calls an upper-half function to read input, the upper-half function extracts characters from the input buffer.

Conceptually, the two halves of a driver only communicate through shared buffers. Upper-half functions place outgoing data in a buffer and extract incoming data from a buffer. The lower half extracts outgoing data from the buffer and sends it to the device, and places incoming data in the buffer. To summarize:

Upper-half functions transfer data between processes and buffers; the lower half transfers data between buffers and the device hardware.

15.6 Synchronization Of Upper Half And Lower Half

In practice, the two halves of the driver usually need to do more than manipulate a shared data structure. For example, an upper-half function may need to start an output transfer if a device is idle. More important, the two halves need to coordinate operations on the request queue and the buffers. For example, if all slots in the output buffer are full when a process tries to write data, the process must be blocked. Later, when characters have been sent to the device and buffer space becomes available, the blocked

†To improve efficiency, most UART hardware has a small on-board character buffer that can hold up to 16 outgoing characters at a time.

process must be allowed to proceed. Similarly, if the input buffer is empty when a process attempts to read from a device, the process must be blocked. Later, when input has been received and placed in the buffer, the process that is waiting for input must be allowed to proceed.

At first glance, synchronization between the upper half and lower half of a driver appears to consist of two instances of *producer–consumer coordination* that can be solved easily with semaphores. On output, the upper-half functions produce data that the lower-half functions consume, and on input, the lower half produces input data that the upper-half functions consume. Input poses no problem for the producer–consumer paradigm; a semaphore can be created that handles coordination. When a process calls an upper-half input function, the process *waits* on the input semaphore until the lower half produces an input data item and *signals* the semaphore.

Output poses an added twist. To understand the problem, recall our restriction on interrupt processing: because it can be executed by the null process, an interrupt function cannot call a function that moves the executing process to any state other than *ready* or *current*. In particular, lower-half routines cannot call *wait*. Consequently, a driver cannot be designed in which a semaphore allows upper-half functions to produce data and lower-half functions to consume data.

How can upper-half and lower-half functions coordinate to control output? Surprisingly, a semaphore solves the problem easily. The trick is to turn around the call to *wait* by changing the purpose of the output semaphore. Instead of having a lower-half routine *wait* for the upper half to produce data, we arrange for the upper half to *wait* for space in the buffer. Thus, we do not view the lower half as a consumer. Instead, a lower-half output function acts as a producer to generate space (i.e., slots) in the buffer, and signals the output semaphore for each slot. To summarize:

> *Semaphores can be used to coordinate the upper half and lower half of a device driver. To avoid having lower-half functions block, output is handled by arranging for upper-half functions to wait for buffer space.*

15.7 UART Hardware FIFOs And Driver Design

The design of the hardware can complicate driver design. For example, consider the UART hardware in the example platforms. The UART device contains two onboard buffers, known as *FIFOs*. One FIFO handles incoming characters, and the other handles outgoing characters. Each FIFO holds up to sixteen characters. The device does not interrupt each time a character arrives. Instead, the hardware generates an interrupt when the first character arrives, but continues to add characters to the input FIFO if they arrive before the interrupt has been serviced. Thus, when it receives an input interrupt, the driver must repeatedly extract characters from the FIFO until the hardware FIFO is empty.

How do multiple input characters affect the driver design? Consider the case where a process is blocked on an input semaphore, waiting for a character to arrive. In theory, once it extracts a character from the device and places the character in the input buffer, the driver should signal the semaphore to reschedule to indicate that a character is available. However, doing so may cause an immediate context switch, leaving additional characters in the FIFO unprocessed. To avoid the problem, our driver uses *resched_cntl* to defer rescheduling temporarily.† After all characters have been extracted from the input FIFO and processed, the driver again calls *resched_cntl* to permit other processes to run.

15.8 The Concept Of A Control Block

We use the term *control block* to refer to the shared data structure associated with a device. More specifically, the control block is associated with a driver for the device, and holds data that the driver uses as well as data needed by the hardware. For example, if a driver uses semaphores to coordinate the upper-half and lower-half functions, the semaphore IDs will be placed in the control block.

Recall that if a system contains multiple copies of a given hardware device, the operating system uses one copy of the device driver code to service all devices. However, each copy of the device must have a separate data structure. That is, the system must have a separate control block for each physical copy of the device. Like most systems, Xinu uses an array to hold *N* copies of the data structure if the system contains *N* copies of a given hardware device. We will see that each copy of the device is assigned a unique index number 0, 1, 2, and so on. The value serves as an index into the array of control blocks. Thus, the copy of a device assigned value i uses the ith entry in the control block array.

When it runs, a device driver function receives an argument that identifies the control block to use. Thus, if a particular system has three serial devices that use the tty abstraction, the operating system contains only one copy of the functions that read and write to a tty device, but contains three separate copies of a tty control block.

15.9 Tty Control Block And Data Declarations

A control block stores information about a specific device, the driver, and the request queues. The control block either contains buffers or contains pointers to buffers in memory.‡ The tty control block also stores IDs of the two semaphores that coordinate input and output. The code in file *tty.h* contains the declaration of the tty control block structure, which is named *ttycblk*. The key components of the *ttycblk* structure consist of the input buffer, *tyibuff*, an output buffer, *tyobuff*, and a separate echo buffer, *tyebuff*. Each buffer used in the tty driver is implemented as an array of characters.

---

†The code for *resched_cntl* can be found in file *resched.c* on page 82.

‡On some systems, I/O buffers must be placed in a special region of memory to permit devices to access the buffers directly.

```
/* tty.h */

#define TY_OBMINSP      20              /* Min space in buffer before   */
                                        /*   processes awakened to write*/
#define TY_EBUFLEN      20              /* Size of echo queue           */

/* Size constants */

#ifndef Ntty
#define Ntty            1               /* Number of serial tty lines   */
#endif
#ifndef TY_IBUFLEN
#define TY_IBUFLEN      128             /* Num. chars in input queue    */
#endif
#ifndef TY_OBUFLEN
#define TY_OBUFLEN      64              /* Num. chars in output queue   */
#endif

/* Mode constants for input and output modes */

#define TY_IMRAW        'R'             /* Raw input mode => no edits   */
#define TY_IMCOOKED     'C'             /* Cooked mode => line editing  */
#define TY_IMCBREAK     'K'             /* Honor echo, etc, no line edit*/
#define TY_OMRAW        'R'             /* Raw output mode => no edits  */

struct  ttycblk {                       /* Tty line control block       */
        char    *tyihead;               /* Next input char to read      */
        char    *tyitail;               /* Next slot for arriving char  */
        char    tyibuff[TY_IBUFLEN];    /* Input buffer (holds one line)*/
        sid32   tyisem;                 /* Input semaphore              */
        char    *tyohead;               /* Next output char to xmit     */
        char    *tyotail;               /* Next slot for outgoing char  */
        char    tyobuff[TY_OBUFLEN];    /* Output buffer                */
        sid32   tyosem;                 /* Output semaphore             */
        char    *tyehead;               /* Next echo char to xmit       */
        char    *tyetail;               /* Next slot to deposit echo ch */
        char    tyebuff[TY_EBUFLEN];    /* Echo buffer                  */
        char    tyimode;                /* Input mode raw/cbreak/cooked */
        bool8   tyiecho;                /* Is input echoed?             */
        bool8   tyieback;               /* Do erasing backspace on echo?*/
        bool8   tyevis;                 /* Echo control chars as ^X ?   */
        bool8   tyecrlf;                /* Echo CR-LF for newline?      */
        bool8   tyicrlf;                /* Map '\r' to '\n' on input?   */
        bool8   tyierase;               /* Honor erase character?       */
        char    tyierasec;              /* Primary erase character      */
```

```
        char    tyierasec2;             /* Alternate erase character    */
        bool8   tyeof;                  /* Honor EOF character?         */
        char    tyeofch;                /* EOF character (usually ^D)   */
        bool8   tyikill;                /* Honor line kill character?   */
        char    tyikillc;               /* Line kill character          */
        int32   tyicursor;              /* Current cursor position      */
        bool8   tyoflow;                /* Honor ostop/ostart?          */
        bool8   tyoheld;                /* Output currently being held? */
        char    tyostop;                /* Character that stops output  */
        char    tyostart;               /* Character that starts output */
        bool8   tyocrlf;                /* Output CR/LF for LF ?        */
        char    tyifullc;               /* Char to send when input full */
};
extern  struct  ttycblk ttytab[];

/* Characters with meaning to the tty driver */

#define TY_BACKSP       '\b'            /* Backspace character          */
#define TY_BACKSP2      '\177'          /* Alternate backspace char.    */
#define TY_BELL         '\07'           /* Character for audible beep   */
#define TY_EOFCH        '\04'           /* Control-D is EOF on input    */
#define TY_BLANK        ' '             /* Blank                        */
#define TY_NEWLINE      '\n'            /* Newline == line feed         */
#define TY_RETURN       '\r'            /* Carriage return character    */
#define TY_STOPCH       '\023'          /* Control-S stops output       */
#define TY_STRTCH       '\021'          /* Control-Q restarts output    */
#define TY_KILLCH       '\025'          /* Control-U is line kill       */
#define TY_UPARROW      '^'             /* Used for control chars (^X)  */
#define TY_FULLCH       TY_BELL         /* Char to echo when buffer full*/

/* Tty control function codes */

#define TC_NEXTC        3               /* Look ahead 1 character       */
#define TC_MODER        4               /* Set input mode to raw        */
#define TC_MODEC        5               /* Set input mode to cooked     */
#define TC_MODEK        6               /* Set input mode to cbreak     */
#define TC_ICHARS       8               /* Return number of input chars */
#define TC_ECHO         9               /* Turn on echo                 */
#define TC_NOECHO       10              /* Turn off echo                */
```

The driver treats each buffer as a circular list, with location zero in an array treated as if it follows the last location. Head and tail pointers give the address of the next location in the array to fill and the next location in the array to empty, respectively. Thus, a programmer can remember an easy rule:

A character is always inserted at the tail and taken from the head, independent of whether a buffer is used for input or output.

Initially, the head and tail each point to location zero, but there is never any confusion about whether an input or output buffer is completely empty or completely full because each buffer has a semaphore that gives the count of characters in the buffer. Semaphore *tyisem* controls the input buffer, and a nonnegative count *n* means the buffer contains *n* characters. Semaphore *tyosem* controls the output buffer, and a nonnegative count *n* means the buffer contains *n* unfilled slots. The echo buffer is an exception. Our design assumes echo is used for a human typing, which means that only a few characters will ever occupy the echo queue. Therefore, we assume that no overflow will occur, which means that no semaphore is needed to control the queue.

15.10 Minor Device Numbers

We said that the configuration program assigns each device in the system a unique device ID. If the system contains multiple physical copies of a device that all use a given abstraction, the IDs assigned to the devices may not be contiguous values. Thus, if a system has three tty devices, the configuration program may assign them device IDs 2, 7, and 8.

We also said that the operating system must allocate one control block per device. For example, if a system contains three tty devices, the system must allocate three copies of the tty control block. Xinu, like many systems, employs a technique that permits efficient access of a control block for a given device. The system assigns a *minor device number* to each copy of the device, and chooses minor device numbers to be contiguous integers starting at zero. Thus, if a system contains three tty devices, they will be assigned minor device numbers 0, 1, and 2. Unlike device IDs, the assignment of minor device numbers is guaranteed to be contiguous.

How does a sequential assignment of minor device numbers make access efficient? We can now understand how device drivers are parameterized, and how they use minor device numbers. A minor device number serves as an index into the array of control blocks for the device. For example, consider how tty control blocks are allocated. As file *tty.h* illustrates, the control blocks are placed in array *ttytab*. The system configuration program defines constant *Ntty* to be the number of tty devices, which is used to declare the size of array *ttytab*. The configuration program assigns each tty device a minor device number starting at 0 and ending at *Ntty*–1. The minor device number is placed in the device switch table entry. Both interrupt-driven routines in the lower half and driver routines in the upper half can access the minor device number. Each function in the upper half has an argument that specifies one of the device table entries. Similarly, when an interrupt occurs, the operating system associates the interrupt with a specific device in the device table. Consequently, each driver function can extract the minor device number, and use the value as an index into array *ttytab*.

15.11 Upper–half Tty Character Input (ttygetc)

Four functions form the foundation for the upper half of the tty driver: *ttygetc*, *ttyputc*, *ttyread*, and *ttywrite*. The functions correspond to the high-level operations *getc*, *putc*, *read*, and *write* described in Chapter 14. The simplest driver routine is *ttygetc*; the code can be found in file *ttygetc.c*.

```
/* ttygetc.c - ttygetc */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttygetc  -  Read one character from a tty device (interrupts disabled)
 *------------------------------------------------------------------------
 */
devcall ttygetc(
          struct dentry *devptr         /* Entry in device switch table */
        )
{
        char    ch;                     /* Character to return          */
        struct  ttycblk *typtr;         /* Pointer to ttytab entry      */

        typtr = &ttytab[devptr->dvminor];

        /* Wait for a character in the buffer and extract one character */

        wait(typtr->tyisem);
        ch = *typtr->tyihead++;

        /* Wrap around to beginning of buffer, if needed */

        if (typtr->tyihead >= &typtr->tyibuff[TY_IBUFLEN]) {
                typtr->tyihead = typtr->tyibuff;
        }

        /* In cooked mode, check for the EOF character */

        if ( (typtr->tyimode == TY_IMCOOKED) && (typtr->tyeof) &&
             (ch == typtr->tyeofch) ) {
                return (devcall)EOF;
        }

        return (devcall)ch;
}
```

When called, *ttygetc* retrieves the minor device number from the device switch table, and then uses the number as an index into array *ttytab* to locate the correct control block. It then executes *wait* on the input semaphore, *tyisem*, blocking until the lower half has deposited a character in the buffer. When *wait* returns, *ttygetc* extracts the next character from the input buffer and updates the head pointer to make it ready for subsequent extractions. Normally, *ttygetc* returns the character to its caller. However, one special case arises: if the driver is honoring *end-of-file* and the character matches the end-of-file character (field *tyeofch* in the control block), *ttygetc* returns constant *EOF*.

15.12 Upper–half Tty Read Function (ttyread)

The *read* operation can be used to obtain multiple characters in a single operation. The driver function that implements the *read* operation, *ttyread*, is shown below in file *ttyread.c*. *Ttyread* is conceptually straightforward: it calls *ttygetc* repeatedly to obtain characters. When the driver is operating in cooked mode, *ttyread* returns a single line of input, stopping after a *NEWLINE* or *RETURN* character; when operating in other modes, *ttyread* reads characters without testing for an end-of-line.

```
/* ttyread.c - ttyread */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttyread  -  Read character(s) from a tty device (interrupts disabled)
 *------------------------------------------------------------------------
 */
devcall ttyread(
          struct dentry *devptr,        /* Entry in device switch table */
          char  *buff,                  /* Buffer of characters         */
          int32 count                   /* Count of character to read   */
        )
{
        struct  ttycblk *typtr;         /* Pointer to tty control block */
        int32   avail;                  /* Characters available in buff.*/
        int32   nread;                  /* Number of characters read    */
        int32   firstch;                /* First input character on line*/
        char    ch;                     /* Next input character         */

        if (count < 0) {
                return SYSERR;
        }
        typtr= &ttytab[devptr->dvminor];
```

```
        if (typtr->tyimode != TY_IMCOOKED) {

                /* For count of zero, return all available characters */

                if (count == 0) {
                        avail = semcount(typtr->tyisem);
                        if (avail == 0) {
                                return 0;
                        } else {
                                count = avail;
                        }
                }
                for (nread = 0; nread < count; nread++) {
                        *buff++ = (char) ttygetc(devptr);
                }
                return nread;
        }

        /* Block until input arrives */

        firstch = ttygetc(devptr);

        /* Check for End-Of-File */

        if (firstch == EOF) {
                return EOF;
        }

        /* Read up to a line */

        ch = (char) firstch;
        *buff++ = ch;
        nread = 1;
        while ( (nread < count) && (ch != TY_NEWLINE) &&
                        (ch != TY_RETURN) ) {
                ch = ttygetc(devptr);
                *buff++ = ch;
                nread++;
        }
        return nread;
}
```

The semantics of how *read* operates on terminals illustrates how the I/O primitives can be adapted to a variety of devices and modes. For example, an application that uses

raw mode may need to read all the characters available from the input buffer without blocking. *Ttyread* cannot simply call *ttygetc* repeatedly because *ttygetc* will block once the buffer is empty. To accommodate non-blocking requests, our driver allows what might otherwise be considered an illegal operation: it interprets a request to *read* zero characters as a request to "read all characters that are waiting."

The code in *ttyread* shows how the zero length requests are handled in raw mode: the driver uses *semcount* to obtain the current count of the input semaphore, *tyisem*. It then knows exactly how many calls can be made to *ttygetc* without blocking.

For cooked mode, the driver blocks until at least one character arrives. It handles the special case of an end-of-file character, and then calls *ttygetc* repeatedly to read the rest of the line.

15.13 Upper–half Tty Character Output (ttyputc)

The upper-half output routines are almost as easy to understand as the upper-half input routines. *Ttyputc* waits for space in the output buffer, deposits the specific character in the output queue, *tyobuff*, and increments the tail pointer, *tyotail*. In addition, *ttyputc* starts output on the device. File *ttyputc.c* contains the code.

```
/* ttyputc.c - ttyputc */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttyputc  -  Write one character to a tty device (interrupts disabled)
 *------------------------------------------------------------------------
 */
devcall ttyputc(
        struct  dentry  *devptr,        /* Entry in device switch table */
        char    ch                      /* Character to write           */
        )
{
        struct  ttycblk *typtr;         /* Pointer to tty control block */

        typtr = &ttytab[devptr->dvminor];

        /* Handle output CRLF by sending CR first */

        if ( ch==TY_NEWLINE && typtr->tyocrlf ) {
                ttyputc(devptr, TY_RETURN);
        }

        wait(typtr->tyosem);            /* Wait for space in queue */
        *typtr->tyotail++ = ch;
```

```
        /* Wrap around to beginning of buffer, if needed */

        if (typtr->tyotail >= &typtr->tyobuff[TY_OBUFLEN]) {
                typtr->tyotail = typtr->tyobuff;
        }

        /* Start output in case device is idle */

        ttykickout((struct uart_csreg *)devptr->dvcsr);

        return OK;
}
```

In addition to the processing mentioned above, *ttyputc* honors one of the tty parameters, *tyocrlf*. When *tyocrlf* is TRUE, each *NEWLINE* should map to the combination *RETURN* plus *NEWLINE*. To write the *RETURN* character, *ttyputc* calls itself recursively.

15.14 Starting Output (ttykickout)

Just before it returns, *ttyputc* calls *ttykickout* to start output. In fact, *ttykickout* does not perform any output to the device because all output is handled by the lower-half function when an output interrupt occurs. To understand how *ttykickout* works, it is necessary to know how an operating system interacts with the device hardware. It may seem that when a character becomes ready for output, *ttyputc* would take the following steps:

```
Interact with the device to determine whether the device is busy;
if (the device is not busy) {
        send the character to the device;
} else {
        instruct the device to interrupt when output finishes;
}
```

Unfortunately, a device operates in parallel with the processor. Therefore, a race condition can occur — between the time the processor obtains the status and the time it instructs the device to interrupt, the device can finish output. If the device is already idle before the processor sends the command, the device will never interrupt.

To avoid the race condition, device hardware is designed to allow an operating system to request an interrupt without testing the device. Making a request is trivial: the driver merely needs to set a bit in one of the device control registers. No race condition occurs because setting the bit causes an interrupt whether the device is currently sending

characters or idle. If the device is busy, the hardware waits until output finishes and the on-board buffer is empty before generating an interrupt; if the device is currently idle when the bit is set, the device interrupts immediately.

Setting the interrupt bit in the device only requires a single assignment statement; the code can be found in file *ttykickout.c*:

```
/* ttykickout.c - ttykickout */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttykickout  -  "Kick" the hardware for a tty device, causing it to
 *                   generate an output interrupt (interrupts disabled)
 *------------------------------------------------------------------------
 */
void    ttykickout(
         struct uart_csreg *csrptr      /* Address of UART's CSRs       */
        )
{
        /* Force the UART hardware generate an output interrupt */

        csrptr->ier = UART_IER_ERBFI | UART_IER_ETBEI;

        return;
}
```

15.15 Upper–half Tty Multiple Character Output (ttywrite)

The *tty* driver also supports multiple-byte output transfers (i.e., *write*s). Driver function *ttywrite*, found in file *ttywrite.c*, handles the output of one or more bytes. *Ttywrite* begins by checking the argument *count*, which specifies the number of bytes to write. A negative count is invalid, and a count of zero is allowed, but means no characters are written.

Once it has finished checking argument *count*, *ttywrite* enters a loop. On each iteration through the loop, *ttywrite* extracts the next character from the user's buffer and calls *ttyputc* to place the character in the output buffer. As we have seen, *ttyputc* will proceed until the output buffer is full, at which time the call to *ttyputc* will block until space becomes available.

```
/* ttywrite.c - ttywrite */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttywrite  -  Write character(s) to a tty device (interrupts disabled)
 *------------------------------------------------------------------------
 */
devcall ttywrite(
          struct dentry *devptr,        /* Entry in device switch table */
          char  *buff,                  /* Buffer of characters         */
          int32 count                   /* Count of character to write  */
        )
{
        /* Handle negative and zero counts */

        if (count < 0) {
                return SYSERR;
        } else if (count == 0){
                return OK;
        }

        /* Write count characters one at a time */

        for (; count>0 ; count--) {
                ttyputc(devptr, *buff++);
        }
        return OK;
}
```

15.16 Lower–half Tty Driver Function (ttyhandler)

The lower half of the tty driver is invoked when an interrupt occurs. The lower half consists of a handler, *ttyhandler*. Recall that a handler is invoked indirectly — an interrupt dispatcher function calls the handler whenever the device interrupts. The dispatch code is trivial, and identical to the dispatch code used with other devices, except for the handler that is invoked. Therefore, we will only examine the handler code.

It is important to understand that the UART hardware uses one interrupt vector for both input and output. That is, the handler will be invoked whenever the device has received one or more incoming characters, or when the device has sent all the characters in its output FIFO and is ready for more. The handler must check a control register in the device to determine whether an input or output interrupt has occurred. File *ttyhandler.c* contains the code.

```
/* ttyhandler.c - ttyhandler */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttyhandler  -  Handle an interrupt for a tty (serial) device
 *------------------------------------------------------------------------
 */
void ttyhandler(void) {
        struct  dentry  *devptr;        /* Address of device control blk*/
        struct  ttycblk *typtr;         /* Pointer to ttytab entry      */
        struct  uart_csreg *csrptr;     /* Address of UART's CSR        */
        byte    iir = 0;                /* Interrupt identification     */
        byte    lsr = 0;                /* Line status                  */


        /* Get CSR address of the device (assume console for now) */

        devptr = (struct dentry *) &devtab[CONSOLE];
        csrptr = (struct uart_csreg *) devptr->dvcsr;

        /* Obtain a pointer to the tty control block */

        typtr = &ttytab[ devptr->dvminor ];

        /* Decode hardware interrupt request from UART device */

        /* Check interrupt identification register */
        iir = csrptr->iir;
        if (iir & UART_IIR_IRQ) {
                return;
        }

        /* Decode the interrupt cause based upon the value extracted    */
        /* from the UART interrupt identification register.  Clear      */
        /* the interrupt source and perform the appropriate handling    */
        /* to coordinate with the upper half of the driver              */

        /* Decode the interrupt cause */

        iir &= UART_IIR_IDMASK;         /* Mask off the interrupt ID */
        switch (iir) {

            /* Receiver line status interrupt (error) */
```

```
            case UART_IIR_RLSI:
                return;

            /* Receiver data available or timed out */

            case UART_IIR_RDA:
            case UART_IIR_RTO:

                resched_cntl(DEFER_START);

                /* While chars avail. in UART buffer, call ttyinter_in  */

                while ( (csrptr->lsr & UART_LSR_DR) != 0) {
                        ttyhandle_in(typtr, csrptr);
                }

                resched_cntl(DEFER_STOP);

                return;

            /* Transmitter output FIFO is empty (i.e., ready for more)  */

            case UART_IIR_THRE:
                ttyhandle_out(typtr, csrptr);
                return;

            /* Modem status change (simply ignore) */

            case UART_IIR_MSC:
                return;
            }
}
```

After obtaining the device's CSR address from the device switch table, *ttyhandler* loads the UART's CSR address into *csrptr*, and then uses *csrptr* to access the UART device. The key step consists of reading the interrupt identification register and using the value to determine the exact reason for the interrupt. The two reasons of interest are an input interrupt (data has arrived) or an output interrupt (i.e., the transmitter FIFO is empty and the driver can send additional characters).

In the code, a *switch* statement chooses among a line status interrupt, an input interrupt, and an output interrupt. In our system, a line status interrupt should not occur, and is considered an error. Therefore, if a line status interrupt does occur, the handler merely returns (i.e., ignores the interrupt).

15.17 Output Interrupt Processing (ttyhandle_out)

Output interrupt processing is the easiest to understand. When an output interrupt occurs, the device has transmitted all characters from the onboard FIFO and is ready for more. *Ttyhandler* clears the interrupt and calls *ttyhandle_out* to restart output. The code for *ttyhandle_out* can be found in file *ttyhandle_out.c*

```
/* ttyhandle_out.c - ttyhandle_out */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttyhandle_out  -  Handle an output on a tty device by sending more
 *                    characters to the device FIFO (interrupts disabled)
 *------------------------------------------------------------------------
 */
void    ttyhandle_out(
         struct ttycblk *typtr,         /* Ptr to ttytab entry          */
         struct uart_csreg *csrptr      /* Address of UART's CSRs       */
        )
{

        int32   ochars;                 /* Number of output chars sent  */
                                        /*   to the UART                */
        int32   avail;                  /* Available chars in output buf*/
        int32   uspace;                 /* Space left in onboard UART   */
                                        /*   output FIFO                */
        byte    ier = 0;

        /* If output is currently held, simply ignore the call */

        if (typtr->tyoheld) {
                return;
        }

        /* If echo and output queues empty, turn off interrupts */

        if ( (typtr->tyehead == typtr->tyetail) &&
             (semcount(typtr->tyosem) >= TY_OBUFLEN) ) {
                ier = csrptr->ier;
                csrptr->ier = ier & ~UART_IER_ETBEI;
                return;
        }

        /* Initialize uspace to the size of the transmit FIFO */
```

```
        uspace = UART_FIFO_SIZE;

        /* While onboard FIFO is not full and the echo queue is */
        /*   nonempty, xmit chars from the echo queue           */

        while ( (uspace>0) &&  typtr->tyehead != typtr->tyetail) {
                csrptr->buffer = *typtr->tyehead++;
                if (typtr->tyehead >= &typtr->tyebuff[TY_EBUFLEN]) {
                        typtr->tyehead = typtr->tyebuff;
                }
                uspace--;
        }

        /* While onboard FIFO is not full and the output queue is       */
        /*   nonempty, transmit chars from the output queue             */

        ochars = 0;
        avail = TY_OBUFLEN - semcount(typtr->tyosem);
        while ( (uspace>0) &&  (avail > 0) ) {
                csrptr->buffer = *typtr->tyohead++;
                if (typtr->tyohead >= &typtr->tyobuff[TY_OBUFLEN]) {
                        typtr->tyohead = typtr->tyobuff;
                }
                avail--;
                uspace--;
                ochars++;
        }
        if (ochars > 0) {
                signaln(typtr->tyosem, ochars);
        }
        return;
}
```

Ttyhandle_out makes a series of tests before starting output. For example, output should not be started if a user has entered control-S. Similarly, there is no need to start output if both the echo and output queues are empty. To understand how *ttyhandle_out* starts output, recall that the underlying hardware has an onboard FIFO that can hold multiple outgoing characters. Once it has determined that output should proceed, *ttyhandle_out* can send up to *UART_FIFO_SIZE* (16) characters to the device. Characters are sent until the FIFO is full or the buffers are empty, whichever occurs first. The echo queue has highest priority. Therefore, *ttyhandle_out* first sends characters from the echo queue. If slots remain in the FIFO, *ttyhandle_out* sends characters from the output queue.

Each time it removes a character from the output queue, *ttyhandle_out* should signal the output semaphore to indicate that another space is available in the buffer. Because a call to *signal* can reschedule, however, *ttyhandle_out* does not call *signal* im-

mediately. Instead, it increments variable *ochars* to count the number of slots being created. Once it has filled the FIFO (or has emptied the output queue), *ttyhandle_out* calls *signaln* to indicate that space is available in the buffer.

15.18 Tty Input Processing (ttyhandle_in)

Input interrupt processing is more complex than output processing because the onboard input FIFO can contain more than one character. Thus, to handle an input interrupt, *ttyhandler*† enters a loop: while the onboard FIFO is not empty, *ttyhandler* calls *ttyhandle_in*, which extracts and processes one character from the UART's input FIFO. To prevent rescheduling until the loop completes and all characters have been extracted from the device, *ttyhandler* uses *resched_cntl*. Thus, although *ttyhandle_in* calls *signal* to make each character available, no rescheduling occurs until all available characters have been extracted from the device.

Processing individual input characters is the most complex part of the tty device driver because it includes code for details such as character echo and line editing. Function *ttyhandle_in* handles the processing for *raw*, *cbreak*, and *cooked* modes. File *ttyhandle_in.c* contains the code.

```
/* ttyhandle_in.c - ttyhandle_in, erase1, eputc, echoch */

#include <xinu.h>

local   void    erase1(struct ttycblk *, struct uart_csreg *);
local   void    echoch(char, struct ttycblk *, struct uart_csreg *);
local   void    eputc(char, struct ttycblk *, struct uart_csreg *);

/*------------------------------------------------------------------------
 *  ttyhandle_in  -  Handle one arriving char (interrupts disabled)
 *------------------------------------------------------------------------
 */
void    ttyhandle_in (
          struct ttycblk *typtr,        /* Pointer to ttytab entry      */
          struct uart_csreg *csrptr     /* Address of UART's CSR        */
        )
{
        char    ch;                     /* Next char from device        */
        int32   avail;                  /* Chars available in buffer    */

        ch = csrptr->buffer;

        /* Compute chars available */
```

†The code for *ttyhandler* can be found on page 304.

```
avail = semcount(typtr->tyisem);
if (avail < 0) {                /* One or more processes waiting*/
        avail = 0;
}

/* Handle raw mode */

if (typtr->tyimode == TY_IMRAW) {
        if (avail >= TY_IBUFLEN) { /* No space => ignore input  */
                return;
        }

        /* Place char in buffer with no editing */

        *typtr->tyitail++ = ch;

        /* Wrap buffer pointer  */

        if (typtr->tyotail >= &typtr->tyobuff[TY_OBUFLEN]) {
                typtr->tyotail = typtr->tyobuff;
        }

        /* Signal input semaphore and return */
        signal(typtr->tyisem);
        return;
}

/* Handle cooked and cbreak modes (common part) */

if ( (ch == TY_RETURN) && typtr->tyicrlf ) {
        ch = TY_NEWLINE;
}

/* If flow control is in effect, handle ^S and ^Q */

if (typtr->tyoflow) {
        if (ch == typtr->tyostart) {        /* ^Q starts output */
                typtr->tyoheld = FALSE;
                ttykickout(csrptr);
                return;
        } else if (ch == typtr->tyostop) {  /* ^S stops output  */
                typtr->tyoheld = TRUE;
                return;
        }
}
```

```
typtr->tyoheld = FALSE;         /* Any other char starts output */

if (typtr->tyimode == TY_IMCBREAK) {       /* Just cbreak mode  */

        /* If input buffer is full, send bell to user */

        if (avail >= TY_IBUFLEN) {
                eputc(typtr->tyifullc, typtr, csrptr);
        } else {        /* Input buffer has space for this char */
                *typtr->tyitail++ = ch;

                /* Wrap around buffer */

                if (typtr->tyitail>=&typtr->tyibuff[TY_IBUFLEN]) {
                        typtr->tyitail = typtr->tyibuff;
                }
                if (typtr->tyiecho) {   /* Are we echoing chars?*/
                        echoch(ch, typtr, csrptr);
                }
        }
        return;

} else {        /* Just cooked mode (see common code above) */

        /* Line kill character arrives - kill entire line */

        if (ch == typtr->tyikillc && typtr->tyikill) {
                typtr->tyitail -= typtr->tyicursor;
                if (typtr->tyitail < typtr->tyibuff) {
                        typtr->tyihead += TY_IBUFLEN;
                }
                typtr->tyicursor = 0;
                eputc(TY_RETURN, typtr, csrptr);
                eputc(TY_NEWLINE, typtr, csrptr);
                return;
        }

        /* Erase (backspace) character */

        if ( ((ch==typtr->tyierasec) || (ch==typtr->tyierasec2))
                                && typtr->tyierase) {
                if (typtr->tyicursor > 0) {
                        typtr->tyicursor--;
                        erase1(typtr, csrptr);
                }
                return;
        }
```

```
/* End of line */

if ( (ch == TY_NEWLINE) || (ch == TY_RETURN) ) {
        if (typtr->tyiecho) {
                echoch(ch, typtr, csrptr);
        }
        *typtr->tyitail++ = ch;
        if (typtr->tyitail>=&typtr->tyibuff[TY_IBUFLEN]) {
                typtr->tyitail = typtr->tyibuff;
        }
        /* Make entire line (plus \n or \r) available */
        signaln(typtr->tyisem, typtr->tyicursor + 1);
        typtr->tyicursor = 0;   /* Reset for next line  */
        return;
}

/* Character to be placed in buffer - send bell if      */
/*      buffer has overflowed                           */

avail = semcount(typtr->tyisem);
if (avail < 0) {
        avail = 0;
}
if ((avail + typtr->tyicursor) >= TY_IBUFLEN-1) {
        eputc(typtr->tyifullc, typtr, csrptr);
        return;
}

/* EOF character: recognize at beginning of line, but   */
/*      print and ignore otherwise.                     */

if (ch == typtr->tyeofch && typtr->tyeof) {
        if (typtr->tyiecho) {
                echoch(ch, typtr, csrptr);
        }
        if (typtr->tyicursor != 0) {
                return;
        }
        *typtr->tyitail++ = ch;
        signal(typtr->tyisem);
        return;
}


/* Echo the character */
```

```
                if (typtr->tyiecho) {
                        echoch(ch, typtr, csrptr);
                }

                /* Insert in the input buffer */

                typtr->tyicursor++;
                *typtr->tyitail++ = ch;

                /* Wrap around if needed */

                if (typtr->tyitail >= &typtr->tyibuff[TY_IBUFLEN]) {
                        typtr->tyitail = typtr->tyibuff;
                }
                return;
        }
}

/*------------------------------------------------------------------------
 *  erase1  -  Erase one character honoring erasing backspace
 *------------------------------------------------------------------------
 */
local   void    erase1(
          struct ttycblk        *typtr, /* Ptr to ttytab entry          */
          struct uart_csreg     *csrptr /* Address of UART's CSRs       */
        )
{
        char    ch;                     /* Character to erase           */

        if ( (--typtr->tyitail) < typtr->tyibuff) {
                typtr->tyitail += TY_IBUFLEN;
        }

        /* Pick up char to erase */

        ch = *typtr->tyitail;
        if (typtr->tyiecho) {                       /* Are we echoing?   */
                if (ch < TY_BLANK || ch == 0177) { /* Nonprintable      */
                        if (typtr->tyevis) {    /* Visual cntl chars */
                                eputc(TY_BACKSP, typtr, csrptr);
                                if (typtr->tyieback) { /* Erase char    */
                                        eputc(TY_BLANK, typtr, csrptr);
                                        eputc(TY_BACKSP, typtr, csrptr);
                                }
                        }
```

```
                        eputc(TY_BACKSP, typtr, csrptr);/* Bypass up arr*/
                        if (typtr->tyieback) {
                                eputc(TY_BLANK, typtr, csrptr);
                                eputc(TY_BACKSP, typtr, csrptr);
                        }
                } else {  /* A normal character that is printable       */
                        eputc(TY_BACKSP, typtr, csrptr);
                        if (typtr->tyieback) {  /* erase the character  */
                                eputc(TY_BLANK, typtr, csrptr);
                                eputc(TY_BACKSP, typtr, csrptr);
                        }
                }
        }
        return;
}

/*------------------------------------------------------------------------
 *  echoch  -  Echo a character with visual and output crlf options
 *------------------------------------------------------------------------
 */
local   void    echoch(
          char  ch,                     /* Character to echo            */
          struct ttycblk *typtr,        /* Ptr to ttytab entry          */
          struct uart_csreg *csrptr     /* Address of UART's CSRs       */
        )
{
        if ((ch==TY_NEWLINE || ch==TY_RETURN) && typtr->tyecrlf) {
                eputc(TY_RETURN, typtr, csrptr);
                eputc(TY_NEWLINE, typtr, csrptr);
        } else if ( (ch<TY_BLANK||ch==0177) && typtr->tyevis) {
                eputc(TY_UPARROW, typtr, csrptr);/* print ^x            */
                eputc(ch+0100, typtr, csrptr);  /* Make it printable    */
        } else {
                eputc(ch, typtr, csrptr);
        }
}

/*------------------------------------------------------------------------
 *  eputc  -  Put one character in the echo queue
 *------------------------------------------------------------------------
 */
local   void    eputc(
          char  ch,                     /* Character to echo            */
          struct ttycblk *typtr,        /* Ptr to ttytab entry          */
          struct uart_csreg *csrptr     /* Address of UART's CSRs       */
        )
```

```
{
        *typtr->tyetail++ = ch;

        /* Wrap around buffer, if needed */

        if (typtr->tyetail >= &typtr->tyebuff[TY_EBUFLEN]) {
                typtr->tyetail = typtr->tyebuff;
        }
        ttykickout(csrptr);
        return;
}
```

15.18.1 Raw Mode Processing

Raw mode is the easiest to understand, and accounts for only a few lines of code. In raw mode, *ttyhandle_in* checks the input buffer to verify that space remains. To do so, it compares the count of the input semaphore (i.e., the number of characters that are currently in the buffer) to the buffer size. If no space remains in the buffer, *ttyhandle_in* merely returns to the handler (i.e., it discards the character). If space remains, *ttyhandle_in* deposits the character at the tail of the input buffer, moves to the next buffer position, signals the input semaphore, and returns.

15.18.2 Cbreak Mode Processing

Cooked and cbreak mode share code that maps *RETURN* to *NEWLINE* and handles output flow control. Field *tyoflow* of the tty control block determines whether the driver currently honors flow control. If it does, the driver suspends output by setting *tyoheld* to *TRUE* when it receives character *tyostop*, and restarts output when it receives character *tyostart*. Characters *tyostart* and *tyostop* are considered "control" characters, so the driver does not place them in the buffer.

Cbreak mode checks the input buffer, and sends character *tyifullc* if the buffer is full. Normally, *tyifullc* is a "bell" that causes the terminal to sound an audible alarm; the idea is that a human who is typing characters will hear the alarm and stop typing until characters have been read and more buffer space becomes available. If the buffer is not full, the code places the character in the buffer, and wraps around the pointer, if necessary. Finally, cbreak mode calls *echoch* to perform character echo.

15.18.3 Cooked Mode Processing

Cooked mode operates like cbreak mode except that it also performs line editing. The driver accumulates lines in the input buffer, using variable *tyicursor* to keep a count of the characters on the "current" line. When the erase character, *tyierasec*, arrives, *ttyhandle_in* decrements *tyicursor* by one, backing up over the previous character, and

calling function *erase1* to erase the character from the display. When the line kill character, *tyikillc*, arrives, *ttyhandle_in* eliminates the current line by setting *tyicursor* to zero and moving the tail pointer back to the beginning of the line. Finally, when a *NEWLINE* or *RETURN* character arrives, *ttyhandle_in* calls *signaln* to make the entire input line available. It resets *tyicursor* to zero for the next line. Note that the test for buffer full always leaves one extra space in the buffer for the end-of-line character (i.e., a *NEWLINE*).

15.19 Tty Control Block Initialization (ttyinit)

Function *ttyinit*, shown below in file *ttyinit.c*, is called once for each tty device. The parameter, *devptr*, is a pointer to the device switch table entry for a tty device. *Ttyinit* extracts the minor device number from the device switch table, uses the number as an index into the *ttytab* array, and sets *typtr* to the tty control block for the device. *Ttyinit* then initializes each field in the control block, setting the device to cooked mode, creating the input and output semaphores, and assigning head and tail pointers to indicate that buffers are empty. After driver parameters and buffers have been initialized, *ttyinit* sets variable *csrptr* to the CSR address of the UART hardware. It then sets the baud rate, sets the bits per character, turns off hardware parity checking, and sets the number of RS-232 stop bits to 1. Finally, *ttyinit* disables transmitter interrupts temporarily.

Once the basic hardware initialization has been completed, *ttyinit* calls *set_evec* to set the interrupt vector to the interrupt function given in the device switch table. After the interrupt vector has been set, *ttyinit* finishes the last steps of hardware initialization: it resets the hardware to permit both transmit and receive interrupts, and calls *ttykickout* to start output.

Ttyinit assumes a tty will be associated with a keyboard and display that a human will use. Consequently, *ttyinit* initializes a tty to cooked mode, and sets parameters assuming a video device that can backspace over characters on the display and erase them. In particular, the parameter *tyieback* causes *ttyhandle_in* to echo three characters, backspace-space-backspace, when it receives the erase character, *tyierasec*. On a display screen, sending the three-character sequence gives the effect of erasing characters as the user backs over them. If you look again at *ttyhandle_in*,† you will see that it carefully backs up the correct number of spaces, even if the user erases a control character that has been displayed as two printable characters.

†The code for *ttyhandle_in* can be found on page 308.

```
/* ttyinit.c - ttyinit */

#include <xinu.h>

struct  ttycblk ttytab[Ntty];

/*------------------------------------------------------------------------
 *  ttyinit  -  Initialize buffers and modes for a tty line
 *------------------------------------------------------------------------
 */
devcall ttyinit(
          struct dentry *devptr         /* Entry in device switch table */
        )
{
        struct  ttycblk *typtr;         /* Pointer to ttytab entry      */
        struct  uart_csreg *uptr;       /* Address of UART's CSRs       */

        typtr = &ttytab[ devptr->dvminor ];

        /* Initialize values in the tty control block */

        typtr->tyihead = typtr->tyitail =       /* Set up input queue   */
                &typtr->tyibuff[0];             /*    as empty          */
        typtr->tyisem = semcreate(0);           /* Input semaphore      */
        typtr->tyohead = typtr->tyotail =       /* Set up output queue  */
                &typtr->tyobuff[0];             /*    as empty          */
        typtr->tyosem = semcreate(TY_OBUFLEN);  /* Output semaphore     */
        typtr->tyehead = typtr->tyetail =       /* Set up echo queue    */
                &typtr->tyebuff[0];             /*    as empty          */
        typtr->tyimode = TY_IMCOOKED;           /* Start in cooked mode */
        typtr->tyiecho = TRUE;                  /* Echo console input   */
        typtr->tyieback = TRUE;                 /* Honor erasing bksp   */
        typtr->tyevis = TRUE;                   /* Visual control chars */
        typtr->tyecrlf = TRUE;                  /* Echo CRLF for NEWLINE*/
        typtr->tyicrlf = TRUE;                  /* Map CR to NEWLINE    */
        typtr->tyierase = TRUE;                 /* Do erasing backspace */
        typtr->tyierasec = TY_BACKSP;           /* Primary erase char   */
        typtr->tyierasec2= TY_BACKSP2;          /* Alternate erase char */
        typtr->tyeof = TRUE;                    /* Honor eof on input   */
        typtr->tyeofch = TY_EOFCH;              /* End-of-file character*/
        typtr->tyikill = TRUE;                  /* Allow line kill      */
        typtr->tyikillc = TY_KILLCH;            /* Set line kill to ^U  */
        typtr->tyicursor = 0;                   /* Start of input line  */
        typtr->tyoflow = TRUE;                  /* Handle flow control  */
        typtr->tyoheld = FALSE;                 /* Output not held      */
```

```
        typtr->tyostop = TY_STOPCH;             /* Stop char is ^S      */
        typtr->tyostart = TY_STRTCH;            /* Start char is ^Q     */
        typtr->tyocrlf = TRUE;                  /* Send CRLF for NEWLINE*/
        typtr->tyifullc = TY_FULLCH;            /* Send ^G when buffer  */
                                                /*   is full            */

        /* Initialize the UART */

        uptr = (struct uart_csreg *)devptr->dvcsr;

        /* Set baud rate */
        uptr->lcr = UART_LCR_DLAB;
        uptr->dlm = 0x00;
        uptr->dll = 0x18;

        uptr->lcr = UART_LCR_8N1;       /* 8 bit char, No Parity, 1 Stop*/
        uptr->fcr = 0x00;               /* Disable FIFO for now         */

        /* Register the interrupt dispatcher for the tty device */

        set_evec( devptr->dvirq, (uint32)devptr->dvintr );

        /* Enable interrupts on the device: reset the transmit and      */
        /*   receive FIFOS, and set the interrupt trigger level         */

        uptr->fcr = UART_FCR_EFIFO | UART_FCR_RRESET |
                        UART_FCR_TRESET | UART_FCR_TRIG2;

        /* Start the device */

        ttykickout(uptr);
        return OK;
}
```

15.20 Device Driver Control (ttycontrol)

So far we have discussed driver functions that handle upper-half data transfer operations (e.g., *read* and *write*), functions that handle lower-half input and output interrupts, and an initialization function that sets parameters at system startup. The I/O interface defined in Chapter 14 provides another type of non-transfer function: *control*. Basically, *control* allows an application to control the device driver or the underlying device. In our example driver, function *ttycontrol*, found in file *ttycontrol.c*, provides basic control functions:

```
/* ttycontrol.c - ttycontrol */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ttycontrol  -  Control a tty device by setting modes
 *------------------------------------------------------------------------
 */
devcall ttycontrol(
          struct dentry *devptr,        /* Entry in device switch table */
          int32  func,                  /* Function to perform          */
          int32  arg1,                  /* Argument 1 for request       */
          int32  arg2                   /* Argument 2 for request       */
        )
{
        struct  ttycblk *typtr;         /* Pointer to tty control block */
        char    ch;                     /* Character for lookahead      */

        typtr = &ttytab[devptr->dvminor];

        /* Process the request */

        switch ( func ) {

        case TC_NEXTC:
                wait(typtr->tyisem);
                ch = *typtr->tyitail;
                signal(typtr->tyisem);
                return (devcall)ch;

        case TC_MODER:
                typtr->tyimode = TY_IMRAW;
                return (devcall)OK;

        case TC_MODEC:
                typtr->tyimode = TY_IMCOOKED;
                return (devcall)OK;

        case TC_MODEK:
                typtr->tyimode = TY_IMCBREAK;
                return (devcall)OK;

        case TC_ICHARS:
                return(semcount(typtr->tyisem));
```

```
        case TC_ECHO:
                typtr->tyiecho = TRUE;
                return (devcall)OK;

        case TC_NOECHO:
                typtr->tyiecho = FALSE;
                return (devcall)OK;

        default:
                return (devcall)SYSERR;
        }
}
```

The control interface for a tty device provides seven *control functions* (i.e., possible operations a process can perform on a tty device). Function *TC_NEXTC* allows an application to "lookahead" (i.e., obtain a copy of the next character waiting to be read without actually reading the character). Three of the control functions (*TC_MODER*, *TC_MODEC*, and *TC_MODEK*) allow a user to set the mode of a tty device to one of the three modes. Functions *TC_ECHO* and *TC_NOECHO* control character echo, allowing a caller to turn off echo, accept input, and then turn echo back on. Finally, function *TC_ICHARS* allows a user to query the driver to determine how many characters are waiting in the input queue.

Observant readers may have noticed that neither parameter *arg1* nor *arg2* is used in function *ttycontrol*. They have been declared, however, because the device-independent I/O routine *control* always provides four arguments when calling a control function, such as *ttycontrol*. Although the compiler cannot perform type-checking on indirect function calls, omitting the argument declarations makes the code less portable and more difficult to understand.

15.21 Perspective

The length of the code in the chapter reveals an important point about device drivers. To understand the point, compare the amount of code used for a trivial serial device to the code used for message passing and process synchronization primitives (i.e., semaphores). Although message passing and semaphores each provide a powerful abstraction, the code is relatively small.

Why does a trivial device driver contain so much code? After all, the driver only makes it possible to read and write characters. The answer lies in the difference between the abstraction the hardware supplies and the abstraction the driver provides. The underlying hardware merely transfers characters, and the output side is independent of the input side. Thus, the hardware does not handle flow control or character echo. Furthermore, the hardware knows nothing about end-of-line translation (i.e., the *crlf* mapping). Consequently, a driver must contain code that handles many details.

Although it may seem complex, the example driver in this chapter is trivial. A production device driver may comprise more than ten thousand lines of source code, and may contain hundreds of functions. Drivers for devices that can be plugged in at runtime (e.g., a USB device) are even more complex than drivers for static devices. Thus, one should appreciate that taken as a whole, code for device drivers is both huge and extremely complex.

15.22 Summary

A device driver consists of a set of functions that control a peripheral hardware device. The driver routines are partitioned into two halves: an upper half that contains the functions called from applications, and a lower half that contains functions that the system calls when a device interrupts. The two halves communicate through a shared data structure called a device control block.

The example device driver examined in the chapter implements a tty abstraction. It manages input and output over serial line hardware, such as the connection to a keyboard. Upper-half functions in the example driver implement *read*, *write*, *getc*, *putc*, and *control* operations. Each upper-half function is called indirectly through the device-switch table. Lower-half functions in the example driver handle interrupts. During an output interrupt, the lower half fills the onboard FIFO from the echo or output queues. During an input interrupt, the lower half extracts and processes characters from the input FIFO.

EXERCISES

15.1 Predict what would happen if two processes executed *ttyread* concurrently when both requested a large number of characters. Experiment and see what happens.

15.2 Find a flaw in *ttyputc* that prevents it from working correctly in raw mode, and repair the flaw. Hint: raw mode means all incoming characters are received without change.

15.3 *Kprintf* uses polled I/O: it disables interrupts, waits until the device is idle, displays its message, and then restores interrupts. What happens if the tty output buffer is full, *kprintf* is called repeatedly, and each call displays a single *NEWLINE* character? Explain.

15.4 The code in *ttyhandle_out* counts characters as they are removed from the buffer, and then calls *signaln* to update the semaphore. Modify the code to use deferred rescheduling. Which version is easier to understand? Which is more efficient? Explain.

15.5 Some systems partition asynchronous device drivers into three levels: interrupt level to do nothing but transfer characters to and from the device, upper level to transfer characters to and from the user, and a middle level to implement a *line discipline* that handles details like character echo, flow control, special processing, and out of band signals. Convert the Xinu tty driver to a three-level scheme, and arrange for a process to execute code in the middle layer.

15.6 Suppose two processes both attempt to use *write()* on the CONSOLE device concurrently. What will the output be? Why?

15.7 Implement a control function that allows a process to obtain exclusive use of a tty device (e.g., the CONSOLE) and another control function that the process can use to release its exclusive use.

15.8 *Ttycontrol* handles changes of mode poorly because it does not reset the cursor or buffer pointers. Rewrite the code to improve it.

15.9 When connecting two computers, it is useful to have flow control in both directions. Modify the tty driver to include a "tandem" mode that sends Control-S when the input buffer is nearly full, and then sends Control-Q when the buffer is half empty.

15.10 When a user changes the mode of a tty device, what should happen to characters already in the input queue (which were accepted before the mode changed)? One possibility is that the queue is discarded. Modify the code to implement discard during a mode change.

Chapter Contents

16.1 Introduction, 325
16.2 Direct Memory Access And Buffers, 325
16.3 Multiple Buffers And Rings, 326
16.4 An Example Ethernet Driver Using DMA, 327
16.5 Device Hardware Definitions And Constants, 328
16.6 Rings And Buffers In Memory, 331
16.7 Definitions Of An Ethernet Control Block, 333
16.8 Device And Driver Initialization, 336
16.9 Reading From An Ethernet Device, 343
16.10 Writing To An Ethernet Device, 347
16.11 Handling Interrupts From An Ethernet Device, 349
16.12 Ethernet Control Functions, 352
16.13 Perspective, 353
16.14 Summary, 354

16

DMA Devices And Drivers (Ethernet)

Modern hardware is stunningly difficult to deal with.

— James Buchanan

16.1 Introduction

Previous chapters on I/O present a general paradigm that uses a device switch table, and explain how a device driver is organized. Chapter 15 shows an example driver for a tty device, and uses the example to illustrate how the upper half and lower half interact.

This chapter extends our discussion of I/O by considering the design of device drivers for hardware devices that can transfer data from or to memory directly. The chapter presents an Ethernet device to show one example of how buffers are organized for such devices and how a device can access the buffers to perform both input and output.

16.2 Direct Memory Access And Buffers

Although a bus can only transfer a word of data at one time, a block-oriented device, such as a disk or a network interface, needs to transfer multiple words of data to fill a given request. The motivation for *Direct Memory Access* (*DMA*) is parallelism: adding intelligence to an I/O device allows the device to perform multiple bus transfers without interrupting the processor. Thus, when they use DMA, a disk device can

transfer an entire disk block between memory and the device before interrupting the processor and a network interface can transfer an entire packet before interrupting the processor.

DMA output is the easiest to understand. As an example, consider DMA output to a disk. To write a disk block, the operating system places the data in a buffer, creates a *write* request in memory, and passes a pointer to the request to the device. Once a request has been passed to the device, the processor is free to continue executing other processes. While the processor executes, the disk DMA hardware uses the bus to access the *write* request, obtain the buffer address, and transfer successive words of data from the buffer to the disk. Once the device has read an entire block of data from memory and written the block to disk, the disk interrupts the processor. If an additional disk block is ready for output, another DMA operation can be started.

DMA input works the other way around. To read a disk block, the operating system allocates a buffer to hold the incoming data, creates a *read* request in memory, and passes the address of the request to the disk device. After initiating the request, the processor is free, and the operating system executes a process that is ready. Simultaneous with processor execution, the DMA hardware uses the bus to access the request, and transfer the block from the disk to the buffer in memory. Once the entire block has been copied into the memory buffer, the disk interrupts the processor. Thus, with DMA, only one interrupt occurs per block transferred.

16.3 Multiple Buffers And Rings

DMA devices are more complex than described above. Instead of passing the device the address of a single request, the hardware requires the operating system to allocate multiple request blocks (each with its own buffer), link them together on a linked list, and pass the address of the list. The device hardware is designed to follow the linked list and perform successive requests without waiting for the processor to restart each operation.† For example, consider a network interface that uses DMA hardware for input. To receive packets from the network, the operating system allocates a linked list of requests, each of which has a buffer that can hold a network packet. The operating system passes the address of the list to the network interface device. When a packet arrives, the network device uses the request to find the address of the buffer, uses DMA to copy the packet into the buffer, generates an interrupt, and automatically moves to the next request without waiting for the processor. As long as requests remain on the list, the device continues to accept incoming packets and place each packet in the buffer associated with the request.

What happens when a DMA device reaches the end of a buffer list? Interestingly, most DMA devices never reach the end of the linked list because the hardware uses a circular linked list, called a *request ring* (or informally, a *buffer ring*). That is, the last node on the list points back to the first. Each node in the list contains two values: a pointer to a buffer and a status bit that tells whether the buffer is ready for use. On input, the operating system initializes each node on the list to point to a buffer and sets

†Some hardware places request blocks in an array and uses an index rather than a pointer, but the concept is the same.

the status to indicate *EMPTY*. When it fills a buffer, the DMA hardware changes the status to *FULL* and generates an interrupt. The device driver function that handles interrupts extracts the data from all buffers that are full, and clears the status bits to indicate that each buffer is *EMPTY*. On the one hand, if the operating system is fast enough, it will be able to extract each incoming packet and mark the buffer *EMPTY* before another packet arrives. Thus, on a fast processor, the DMA hardware will keep moving around the ring without ever encountering a buffer that is marked *FULL*. On the other hand, if the operating system cannot process packets as fast as they arrive, the device will eventually fill all the buffers and will encounter a buffer marked *FULL*. If it travels completely around the ring and encounters a full buffer, the DMA hardware sets an error indicator (typically an *overflow* bit) and generates an interrupt to inform the operating system.

Most DMA hardware also uses a circular linked list for output. The operating system creates the ring with each buffer marked *EMPTY*. When it has a packet to send, the operating system places the packet in the next available output buffer, marks the buffer *FULL*, and starts the device if the device is not currently running. The device moves to the next buffer, uses DMA to obtain a copy of the packet from the buffer, and transmits the packet over the network. Once started, the DMA output hardware continues to move around the ring until it reaches an empty buffer. Thus, if applications generate data fast enough, the DMA hardware will transmit packets continuously without ever encountering an empty buffer.

16.4 An Example Ethernet Driver Using DMA

An example will clarify the discussion above. Our example driver is written for the Ethernet device on the Quark SoC in the Galileo. Although many of the details are specific to the Ethernet on the Galileo board, the interaction between the processor and device is typical of most DMA devices. For example, the code for the Ethernet device in the BeagleBone Black is similar.

The Quark Ethernet handles both input and output, and has a separate DMA engine for each. That is, a driver must create two rings — one ring has buffers used to receive packets, and the other has buffers used to send packets. The device has separate registers that a driver uses to pass pointers to the input and output rings, and the hardware allows input and output to proceed simultaneously. Despite operating independently, both input and output interrupts use a single interrupt vector. Therefore, the driver acts like the tty driver in Chapter 15: when an interrupt occurs, the handler interacts with the device to determine whether the interrupt corresponds to an input or output operation, and proceeds accordingly.

16.5 Device Hardware Definitions And Constants

File *quark_eth.h* defines constants and structures for the Ethernet hardware. The file contains many details and may seem confusing. For now, it is sufficient to know that the definitions are taken directly from the vendor's manual for the device. For example, struct *eth_q_csreg* specifies the format of control and status registers as specified by the hardware vendor.

We will see that the transmit and receive rings each consist of a circular linked list of *descriptors*, where a descriptor contains status information, a pointer to a buffer in memory, and a pointer to the next descriptor in the ring. For now, it is sufficient to know that structures *eth_q_tx_desc* and *eth_q_rx_desc* define the descriptors that the DMA expects to find in the transmit and receive rings, respectively.

```
/* quark_eth.h */

/* Definitions for Intel Quark Ethernet */

#define INTEL_ETH_QUARK_PCI_DID 0x0937          /* MAC PCI Device ID    */
#define INTEL_ETH_QUARK_PCI_VID 0x8086          /* MAC PCI Vendor ID    */

struct eth_q_csreg {
        uint32  maccr;          /* MAC Configuration Register           */
        uint32  macff;          /* MAC Frame Filter Register            */
        uint32  hthr;           /* Hash Table High Register             */
        uint32  htlr;           /* Hash Table Low Register              */
        uint32  gmiiar;         /* GMII Address Register                */
        uint32  gmiidr;         /* GMII Data Register                   */
        uint32  fcr;            /* Flow Control Register                */
        uint32  vlantag;        /* VLAV Tag Register                    */
        uint32  version;        /* Version Register                     */
        uint32  debug;          /* Debug Register                       */
        uint32  res1[4];        /* Skipped Addresses                    */
        uint32  ir;             /* Interrupt Register                   */
        uint32  imr;            /* Interrupt Mask Register              */
        uint32  macaddr0h;      /* MAC Address0 High Register           */
        uint32  macaddr0l;      /* MAC Address0 Low Register            */
        uint32  res2[46];
        uint32  mmccr;          /* MAC Management Counter Cntl Register */
        uint32  mmcrvcir;       /* MMC Receive Interrupt Register       */
        uint32  mmctxir;        /* MMC Transmit Interrupt Register      */
        uint32  res3[957];      /* Skipped Addresses                    */
        uint32  bmr;            /* Bus Mode Register                    */
        uint32  tpdr;           /* Transmit Poll Demand Register        */
        uint32  rpdr;           /* Receive Poll Demand Register         */
```

```
        uint32  rdla;           /* Receive Descriptor List Addr         */
        uint32  tdla;           /* Transmit Descriptor List Addr        */
        uint32  sr;             /* Status Register                      */
        uint32  omr;            /* Operation Mode Register              */
        uint32  ier;            /* Interrupt Enable Register            */
};

/* Individual Bits in Control and Status Registers */

/* MAC Configuration Register   */

#define ETH_QUARK_MACCR_PE2K    0x08000000      /* Enable 2K Packets    */
#define ETH_QUARK_MACCR_WD      0x00800000      /* Watchdog Disable     */
#define ETH_QUARK_MACCR_JD      0x00400000      /* Jabber Disable       */
#define ETH_QUARK_MACCR_JE      0x00100000      /* Jumbo Frame Enable   */

/* Inter-frame gap values */
#define ETH_QUARK_MACCR_IFG96   0x00000000      /* 96 bit times         */
#define ETH_QUARK_MACCR_IFG88   0x00020000      /* 88 bit times         */
#define ETH_QUARK_MACCR_IFG80   0x00040000      /* 80 bit times         */
#define ETH_QUARK_MACCR_IFG40   0x000E0000      /* 40 bit times         */
#define ETH_QUARK_MACCR_IFG64   0x00080000      /* 64 bit times         */

#define ETH_QUARK_MACCR_DCRS    0x00010000      /* Dis. C. Sense dur TX */
#define ETH_QUARK_MACCR_RMIISPD10  0x00000000   /* RMII Speed = 10 Mbps */
#define ETH_QUARK_MACCR_RMIISPD100 0x00004000   /* RMII Speed = 100 Mbps*/
#define ETH_QUARK_MACCR_DO      0x00002000      /* Disable Receive Own  */
#define ETH_QUARK_MACCR_LM      0x00001000      /* Loopback Mode Enable */
#define ETH_QUARK_MACCR_DM      0x00000800      /* Duplex Mode Enable   */
#define ETH_QUARK_MACCR_IPC     0x00000400      /* Checksum Offload     */
#define ETH_QUARK_MACCR_DR      0x00000200      /* Disable Retry        */
#define ETH_QUARK_MACCR_ACS     0x00000080      /* Auto Pad or CRC Strip*/
#define ETH_QUARK_MACCR_DC      0x00000010      /* Deferral Check       */
#define ETH_QUARK_MACCR_TE      0x00000008      /* Transmitter Enable   */
#define ETH_QUARK_MACCR_RE      0x00000004      /* Receiver Enable      */
#define ETH_QUARK_MACCR_PRELEN7 0x00000000      /* Preamble = 7 bytes   */
#define ETH_QUARK_MACCR_PRELEN5 0x00000001      /* Preamble = 5 bytes   */
#define ETH_QUARK_MACCR_PRELEN3 0x00000002      /* Preamble = 3 bytes   */

#define ETH_QUARK_MMC_CNTFREEZ 0x00000008   /* Freeze MMC counter values*/
#define ETH_QUARK_MMC_CNTRST   0x00000001   /* Reset all cntrs to zero  */

/* GMII Address Register        */
#define ETH_QUARK_GMIIAR_PAMASK 0x0000F800      /* Phys Layer Addr Mask */
#define ETH_QUARK_GMIIAR_GRMASK 0x000007C0      /* GMII Register Mask   */
```

```
#define ETH_QUARK_GMIIAR_CR     0x00000004      /* Clk Range = 100-150  */
                                                /*    MHz for Quark     */
#define ETH_QUARK_GMIIAR_GW     0x00000002      /* GMII Write Enable    */
#define ETH_QUARK_GMIIAR_GB     0x00000001      /* GMII Busy            */

/* Bus Mode Register */
#define ETH_QUARK_BMR_SWR       0x00000001      /* Software Reset       */

/* Status Register */
#define ETH_QUARK_SR_MMCI       0x08000000      /* MAC MMC interrupt    */
#define ETH_QUARK_SR_TS_SUSP    0x00600000      /* TX DMA is suspended  */
#define ETH_QUARK_SR_NIS        0x00010000      /* Normal Int summary   */
#define ETH_QUARK_SR_AIS        0x00008000      /* Abnorm Intrupt summ. */
#define ETH_QUARK_SR_RI         0x00000040      /* Receive Interrupt    */
#define ETH_QUARK_SR_TI         0x00000001      /* Transmit Interrupt   */

/* Operation Mode Register */
#define ETH_QUARK_OMR_TSF       0x00200000      /* Tx store and forward */
#define ETH_QUARK_OMR_ST        0x00002000      /* Start/Stop TX        */
#define ETH_QUARK_OMR_SR        0x00000002      /* Start/Stop RX        */

/* Interrupt Enable Register */
#define ETH_QUARK_IER_NIE       0x00010000      /* Enable Norm Int Summ.*/
#define ETH_QUARK_IER_AIE       0x00008000      /* Enable Abnnom "   "  */
#define ETH_QUARK_IER_RIE       0x00000040      /* Enable RX Interrupt  */
#define ETH_QUARK_IER_TIE       0x00000001      /* Enable TX Interrupt  */

/* Quark Ethernet Transmit Descriptor */

struct eth_q_tx_desc {
        uint32  ctrlstat;       /* Control and status   */
        uint16  buf1size;       /* Size of buffer 1     */
        uint16  buf2size;       /* Size of buffer 2     */
        uint32  buffer1;        /* Address of buffer 1  */
        uint32  buffer2;        /* Address of buffer 2  */
};

#define ETH_QUARK_TDCS_OWN      0x80000000      /* Descrip. owned by DMA*/
#define ETH_QUARK_TDCS_IC       0x40000000      /* Int on Completion    */
#define ETH_QUARK_TDCS_LS       0x20000000      /* Last Segment         */
#define ETH_QUARK_TDCS_FS       0x10000000      /* First Segment        */
#define ETH_QUARK_TDCS_TER      0x00200000      /* Transmit End of Ring */
#define ETH_QUARK_TDCS_ES       0x00008000      /* Error Summary        */

/* Quark Ethernet Receive Descriptor */
```

```
struct eth_q_rx_desc {
        uint32  status;         /* Desc status word     */
        uint16  buf1size;       /* Size of buffer 1     */
        uint16  buf2size;       /* Size of buffer 2     */
        uint32  buffer1;        /* Address of buffer 1  */
        uint32  buffer2;        /* Address of buffer 2  */
};
#define rdctl1  buf1size        /* Buffer 1 size field has control bits too */
#define rdctl2  buf2size        /* Buffer 2 size field has control bits too */

#define ETH_QUARK_RDST_OWN      0x80000000      /* Descrip. owned by DMA*/
#define ETH_QUARK_RDST_ES       0x00008000      /* Error Summary        */
#define ETH_QUARK_RDST_FS       0x00000200      /* First Segment        */
#define ETH_QUARK_RDST_LS       0x00000100      /* Last segment         */
#define ETH_QUARK_RDST_FTETH    0x00000020      /* Frame Type = Ethernet*/

#define ETH_QUARK_RDCTL1_DIC    0x8000  /* Dis. Int on Complet. */
#define ETH_QUARK_RDCTL1_RER    0x8000  /* Recv End of Ring     */

#define ETH_QUARK_RX_RING_SIZE  32
#define ETH_QUARK_TX_RING_SIZE  16

#define ETH_QUARK_INIT_DELAY    500000          /* Delay in micro secs  */
#define ETH_QUARK_MAX_RETRIES   3               /* Max retries for init */
```

16.6 Rings And Buffers In Memory

From a device's perspective, an input or output ring consists of a linked list of descriptors in memory. We said that each descriptor on a ring contains a status word that specifies whether the associated buffer is empty or full. The descriptor also contains a pointer to a buffer in memory and a pointer to the next descriptor on the list. Figure 16.1 illustrates the conceptual organization of transmit and receive rings, and shows that each descriptor contains a pointer to a buffer as well as a pointer to the next ring.

Figure 16.1 Illustration of the transmit and receive rings used with the example DMA hardware device.

As the figure illustrates, the descriptors are arranged in a circular linked list where the final node points back to the first node. When reading the code, it will be important to remember that the Ethernet device views each ring as a linked list. Thus, the DMA hardware merely follows the pointer in one node to get to the next node on the list. The reason the distinction is important arises from the way our driver code allocates storage. The driver uses *getmem* to allocate enough memory for all descriptors in a ring, and then links them together. Because they are contiguous, driver code can use pointer increment to move along the set of nodes. Figure 16.2 shows the structure of a node and illustrates how nodes of a ring are stored in contiguous storage.

Figure 16.2 (a) The three fields in a node on a receive or transmit ring, and (b) a ring stored in contiguous memory with the second node shaded.

16.7 Definitions Of An Ethernet Control Block

File *ether.h* defines constants and data structures used by the Ethernet driver, including the format of an Ethernet packet header, the layout of a packet buffer in memory, and the contents of an Ethernet control block.

```
/* ether.h */

/* Ethernet packet format:

 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |  Dest. MAC (6)  |  Src. MAC (6)   |Type (2)|      Data (46-1500)...   |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
*/

#define ETH_ADDR_LEN    6               /* Len. of Ethernet (MAC) addr. */
typedef unsigned char   Eaddr[ETH_ADDR_LEN];/* Physical Ethernet address*/

/* Ethernet packet header */

struct  etherPkt {
        byte    dst[ETH_ADDR_LEN];      /* Destination Mac address      */
        byte    src[ETH_ADDR_LEN];      /* Source Mac address           */
        uint16  type;                   /* Ether type field             */
        byte    data[1];                /* Packet payload               */
};

#define ETH_HDR_LEN             14      /* Length of Ethernet packet    */
                                        /*   header                     */

/* Ethernet DMA buffer sizes */

#define ETH_MTU                 1500    /* Maximum transmission unit    */
#define ETH_VLAN_LEN            4       /* Length of Ethernet vlan tag  */
#define ETH_CRC_LEN             4       /* Length of CRC on Ethernet    */
                                        /*   frame                      */

#define ETH_MAX_PKT_LEN ( ETH_HDR_LEN + ETH_VLAN_LEN + ETH_MTU )

#define ETH_BUF_SIZE            2048    /* A multiple of 16 greater     */
                                        /*   than the max packet        */
                                        /*   size (for cache alignment) */

/* State of the Ethernet interface */

#define ETH_STATE_FREE          0       /* Control block is unused      */
#define ETH_STATE_DOWN          1       /* Interface is inactive        */
#define ETH_STATE_UP            2       /* Interface is currently active*/

/* Ethernet device control functions */
```

```
#define ETH_CTRL_GET_MAC        1       /* Get the MAC for this device  */
#define ETH_CTRL_ADD_MCAST      2       /* Add a multicast address      */
#define ETH_CTRL_REMOVE_MCAST   3       /* Remove a multicast address   */

/* Ethernet multicast */

#define ETH_NUM_MCAST           32      /* Max multicast addresses      */

/* Ehternet NIC type */

#define ETH_TYPE_3C905C         1       /* 3COM 905C                    */
#define ETH_TYPE_E1000E         2       /* Intel E1000E                 */
#define ETH_TYPE_QUARK_ETH      3       /* Ethernet on Quark board      */

/* Control block for Ethernet device */

struct  ethcblk {
        byte    state;          /* ETH_STATE_... as defined above       */
        struct  dentry  *phy;   /* physical eth device for Tx DMA       */
        byte    type;           /* NIC type_... as defined above        */

        /* Pointers to associated structures */

        struct  dentry  *dev;   /* Address in device switch table       */
        void    *csr;           /* Control and status regsiter address  */
        uint32  pcidev;         /* PCI device number                    */
        uint32  iobase;         /* I/O base from config                 */
        uint32  flashbase;      /* Flash base from config               */
        uint32  membase;        /* Memory base for device from config   */

        void    *rxRing;        /* Ptr to array of recv ring descriptors*/
        void    *rxBufs;        /* Ptr to Rx packet buffers in memory   */
        uint32  rxHead;         /* Index of current head of Rx ring     */
        uint32  rxTail;         /* Index of current tail of Rx ring     */
        uint32  rxRingSize;     /* Size of Rx ring descriptor array     */
        uint32  rxIrq;          /* Count of Rx interrupt requests       */

        void    *txRing;        /* Ptr to array of xmit ring descriptors*/
        void    *txBufs;        /* Ptr to Tx packet buffers in memory   */
        uint32  txHead;         /* Index of current head of Tx ring     */
        uint32  txTail;         /* Index of current tail of Tx ring     */
        uint32  txRingSize;     /* Size of Tx ring descriptor array     */
        uint32  txIrq;          /* Count of Tx interrupt requests       */

        uint8   devAddress[ETH_ADDR_LEN];/* MAC address                 */
```

```
        uint8   addrLen;        /* Hardware address length              */
        uint16  mtu;            /* Maximum transmission unit (payload)  */

        uint32  errors;         /* Number of Ethernet errors            */
        sid32   isem;           /* Semaphore for Ethernet input         */
        sid32   osem;           /* Semaphore for Ethernet output        */
        uint16  istart;         /* Index of next packet in the ring     */

        int16   inPool;         /* Buffer pool ID for input buffers     */
        int16   outPool;        /* Buffer pool ID for output buffers    */

        int16   proms;          /* Nonzero => promiscuous mode          */

        int16   ed_mcset;       /* Nonzero => multicast reception set   */
        int16   ed_mcc;         /* Count of multicast addresses         */
        Eaddr   ed_mca[ETH_NUM_MCAST];/* Array of multicast addrs       */
};

extern  struct  ethcblk ethertab[];     /* Array of control blocks      */
```

16.8 Device And Driver Initialization

Recall that the operating calls function *init* on each device at startup. The device switch table is configured such that a call to *init* on the Ethernet device invokes function *ethinit*, which initializes both the Ethernet device and the device driver data structures. File *ethinit.c* contains the code.

```
/* ethinit.c - ethinit, eth_phy_read, eth_phy_write */

#include <xinu.h>

struct  ethcblk ethertab[1];

/*------------------------------------------------------------------------
 * eth_phy_read  -  Read a PHY register
 *------------------------------------------------------------------------
 */
uint16  eth_phy_read    (
          volatile      struct eth_q_csreg *csrptr,     /* CSR address  */
          uint32        regnum                          /* Register     */
        )
{
        uint32  retries;                /* No. of retries for read      */
```

```
        /* Wait for the MII to be ready */

        while(csrptr->gmiiar & ETH_QUARK_GMIIAR_GB);

        /* Prepare the GMII address register for read transaction */

        csrptr->gmiiar =
                (1 << 11)               | /* Physical Layer Address = 1 */
                (regnum << 6)           | /* PHY Register Number        */
                (ETH_QUARK_GMIIAR_CR)   | /* GMII Clock Range 100-150MHz*/
                (ETH_QUARK_GMIIAR_GB);    /* Start the transaction      */

        /* Wait for the transaction to complete */

        retries = 0;
        while(csrptr->gmiiar & ETH_QUARK_GMIIAR_GB) {
                DELAY(ETH_QUARK_INIT_DELAY);
                if((++retries) > ETH_QUARK_MAX_RETRIES)
                        return 0;
        }

        /* Transaction is complete, read the PHY register value from    */
        /*   the GMII data register                                     */
        return (uint16)csrptr->gmiidr;
}

/*------------------------------------------------------------------------
 * eth_phy_write  -  Write a PHY register
 *------------------------------------------------------------------------
 */
void    eth_phy_write   (
          volatile      struct eth_q_csreg *csrptr, /* CSR address      */
          uint32        regnum,                     /* Register         */
                        uint16  value               /* Value to write   */
        )
{
        uint32  retries; /* No. of retries for write */

        /* Wait for the MII to be ready */

        while(csrptr->gmiiar & ETH_QUARK_GMIIAR_GB);

        /* Write the value to GMII data register */

        csrptr->gmiidr = (uint32)value;
```

```
        /* Prepare the GMII address register for write transaction */

        csrptr->gmiiar =
                (1 << 11)               | /* Physical Layer Address = 1 */
                (regnum << 6)           | /* PHY Register Number        */
                (ETH_QUARK_GMIIAR_CR)   | /* GMII Clock Range 100-150MHz*/
                (ETH_QUARK_GMIIAR_GW)   | /* Write transaction          */
                (ETH_QUARK_GMIIAR_GB);    /* Start the transaction      */

        /* Wait till the transaction is complete */

        retries = 0;
        while(csrptr->gmiiar & ETH_QUARK_GMIIAR_GB) {
                DELAY(ETH_QUARK_INIT_DELAY);
                if((++retries) > ETH_QUARK_MAX_RETRIES)
                        return;
        }
}

/*------------------------------------------------------------------------
 * eth_phy_reset  -  Reset an Ethernet PHY
 *------------------------------------------------------------------------
 */
int32   eth_phy_reset   (
          volatile struct eth_q_csreg *csrptr   /* CSR address          */
        )
{
        uint16  value;          /* Variable to read in PHY registers    */
        uint32  retries;        /* No.  of retries for reset            */

        /* Read the PHY control register (register 0) */

        value = eth_phy_read(csrptr, 0);

        /* Set bit 15 in control register to reset the PHY */

        eth_phy_write(csrptr, 0, (value | 0x8000));

        /* Wait for PHY reset process to complete */

        retries = 0;
        while(eth_phy_read(csrptr, 0) & 0x8000) {
                DELAY(ETH_QUARK_INIT_DELAY);
                if((++retries) > ETH_QUARK_MAX_RETRIES)
                        return SYSERR;
        }
```

```
        /* See if the PHY has auto-negotiation capability */

        value = eth_phy_read(csrptr, 1);        /* PHY Status register  */
        if(value & 0x0008) { /* Auto-negotiation capable */

                /* Wait for the auto-negotiation process to complete */

                retries = 0;
                while((eth_phy_read(csrptr, 1) & 0x0020) == 0) {
                        DELAY(ETH_QUARK_INIT_DELAY);
                        if((++retries) > ETH_QUARK_MAX_RETRIES)
                                return SYSERR;
                }
        }

        /* Wait for the Link to be Up */

        retries = 0;
        while((eth_phy_read(csrptr, 1) & 0x0004) == 0) {
                DELAY(ETH_QUARK_INIT_DELAY);
                if((++retries) > ETH_QUARK_MAX_RETRIES)
                        return SYSERR;
        }

        DELAY(ETH_QUARK_INIT_DELAY);

        kprintf("Ethernet Link is Up\n");

        return OK;
}

/*------------------------------------------------------------------------
 * ethinit  -  Initialize the Intel Quark Ethernet device
 *------------------------------------------------------------------------
 */
int32   ethinit (
          struct dentry *devptr         /* Entry in device switch table */
        )
{
        struct  ethcblk *ethptr;                /* Ptr to control block */
        volatile struct eth_q_csreg *csrptr;    /* Ptr to CSR           */
        struct  eth_q_tx_desc *tx_descs;        /* Array of tx descs    */
        struct  eth_q_rx_desc *rx_descs;        /* Array of rx descs    */
        struct  netpacket *pktptr;              /* Pointer to a packet  */
        void    *temptr;                        /* Temp. pointer        */
```

```
uint32  retries;                        /* Retry count for reset*/
int32   retval;
int32   i;

ethptr = &ethertab[devptr->dvminor];

ethptr->csr = (struct eth_q_csreg *)devptr->dvcsr;
csrptr = (struct eth_q_csreg *)ethptr->csr;

/* Enable CSR Memory Space, Enable Bus Master */
pci_write_config_word(ethptr->pcidev, 0x4, 0x0006);

/* Reset the PHY */
retval = eth_phy_reset(csrptr);
if(retval == SYSERR) {
        return SYSERR;
}

/* Reset the Ethernet MAC */
csrptr->bmr |= ETH_QUARK_BMR_SWR;

/* Wait for the MAC Reset process to complete */
retries = 0;
while(csrptr->bmr & ETH_QUARK_BMR_SWR) {
        DELAY(ETH_QUARK_INIT_DELAY);
        if((++retries) > ETH_QUARK_MAX_RETRIES)
                return SYSERR;
}

/* Transmit Store and Forward */
csrptr->omr |= ETH_QUARK_OMR_TSF;

/* Set the interrupt handler */
set_evec(devptr->dvirq, (uint32)devptr->dvintr);

/* Set the MAC Speed = 100Mbps, Full Duplex mode */
csrptr->maccr |= (ETH_QUARK_MACCR_RMIISPD100 |
                  ETH_QUARK_MACCR_DM);

/* Reset the MMC Counters */
csrptr->mmccr |= ETH_QUARK_MMC_CNTFREEZ | ETH_QUARK_MMC_CNTRST;

/* Retrieve the MAC address from SPI flash */
get_quark_pdat_entry_data_by_id(QUARK_MAC1_ID,
                (char*)(ethptr->devAddress), ETH_ADDR_LEN);
```

```
kprintf("MAC address is %02x:%02x:%02x:%02x:%02x:%02x\n",
        0xff&ethptr->devAddress[0],
        0xff&ethptr->devAddress[1],
        0xff&ethptr->devAddress[2],
        0xff&ethptr->devAddress[3],
        0xff&ethptr->devAddress[4],
        0xff&ethptr->devAddress[5]);

/* Add the MAC address read from SPI flash into the     */
/* macaddr registers for address filtering              */
csrptr->macaddr0l = (uint32)(*((uint32 *)ethptr->devAddress));
csrptr->macaddr0h = ((uint32)
        (*((uint16 *)(ethptr->devAddress + 4))) | 0x80000000);

ethptr->txRingSize = ETH_QUARK_TX_RING_SIZE;

/* Allocate memory for the transmit ring */
temptr = (void *)getmem(sizeof(struct eth_q_tx_desc) *
                                (ethptr->txRingSize+1));
if((int)temptr == SYSERR) {
        return SYSERR;
}
memset(temptr, 0, sizeof(struct eth_q_tx_desc) *
                                (ethptr->txRingSize+1));

/* The transmit descriptors need to be 4-byte aligned */
ethptr->txRing = (void *)(((uint32)temptr + 3) & (~3));

/* Allocate memory for transmit buffers */
ethptr->txBufs = (void *)getmem(sizeof(struct netpacket) *
                                (ethptr->txRingSize+1));
if((int)ethptr->txBufs == SYSERR) {
        return SYSERR;
}
ethptr->txBufs = (void *)(((uint32)ethptr->txBufs + 3) & (~3));

/* Pointers to initialize transmit descriptors */
tx_descs = (struct eth_q_tx_desc *)ethptr->txRing;
pktptr = (struct netpacket *)ethptr->txBufs;

/* Initialize the transmit descriptors */
for(i = 0; i < ethptr->txRingSize; i++) {
        tx_descs[i].buffer1 = (uint32)(pktptr + i);
}
```

```
/* Create the output synchronization semaphore */
ethptr->osem = semcreate(ethptr->txRingSize);
if((int)ethptr->osem == SYSERR) {
        return SYSERR;
}

ethptr->rxRingSize = ETH_QUARK_RX_RING_SIZE;

/* Allocate memory for the receive descriptors */
temptr = (void *)getmem(sizeof(struct eth_q_rx_desc) *
                                (ethptr->rxRingSize+1));
if((int)temptr == SYSERR) {
        return SYSERR;
}
memset(temptr, 0, sizeof(struct eth_q_rx_desc) *
                                (ethptr->rxRingSize+1));

/* Receive descriptors must be 4-byte aligned */
ethptr->rxRing = (struct eth_q_rx_desc *)
                                (((uint32)temptr + 3) & (~3));

/* Allocate memory for the receive buffers */
ethptr->rxBufs = (void *)getmem(sizeof(struct netpacket) *
                                        (ethptr->rxRingSize+1));
if((int)ethptr->rxBufs == SYSERR) {
        return SYSERR;
}

/* Receive buffers must be 4-byte aligned */
ethptr->rxBufs = (void *)(((uint32)ethptr->rxBufs + 3) & (~3));

/* Pointer to initialize receive descriptors */
rx_descs = (struct eth_q_rx_desc *)ethptr->rxRing;

/* Pointer to data buffers */
pktptr = (struct netpacket *)ethptr->rxBufs;

/* Initialize the receive descriptors */
for(i = 0; i < ethptr->rxRingSize; i++) {

        rx_descs[i].status   = ETH_QUARK_RDST_OWN;
        rx_descs[i].buf1size = (uint32)sizeof(struct netpacket);
        rx_descs[i].buffer1  = (uint32)(pktptr + i);
}
```

```
        /* Indicate end of ring on last descriptor */
        rx_descs[ethptr->rxRingSize-1].buf1size |= (ETH_QUARK_RDCTL1_RER);

        /* Create the input synchronization semaphore */
        ethptr->isem = semcreate(0);
        if((int)ethptr->isem == SYSERR) {
                return SYSERR;
        }

        /* Enable the Transmit and Receive Interrupts */
        csrptr->ier = ( ETH_QUARK_IER_NIE |
                        ETH_QUARK_IER_TIE |
                        ETH_QUARK_IER_RIE );

        /* Initialize the transmit descriptor base address */
        csrptr->tdla = (uint32)ethptr->txRing;

        /* Initialize the receive descriptor base address */
        csrptr->rdla = (uint32)ethptr->rxRing;

        /* Enable the MAC Receiver and Transmitter */
        csrptr->maccr |= (ETH_QUARK_MACCR_TE | ETH_QUARK_MACCR_RE);

        /* Start the Transmit and Receive Processes in the DMA */
        csrptr->omr |= (ETH_QUARK_OMR_ST | ETH_QUARK_OMR_SR);

        return OK;

}
```

When it is called, *ethinit* initializes fields in the device control block, and then initializes the hardware. Many of the details depend on the specific Ethernet hardware, but most hardware has equivalent functionality. Once the descriptor rings have been initialized, *ethinit* enables transmit and receiver interrupts in the hardware, making the device ready to transmit and receive packets.

16.9 Reading From An Ethernet Device

Because the DMA engine uses the input ring to store incoming packets in successive buffers, reading from an Ethernet device does not involve much interaction with the device hardware. Instead, the driver uses a semaphore to coordinate reading: the semaphore begins with a count of zero, and is signalled each time a packet arrives.

When it reads a packet, a process waits on the semaphore, then extracts a packet from the next ring buffer. If no packet is available when a process attempts to read a packet, the caller will be blocked. Once a packet becomes available and the interrupt handler signals the semaphore, the caller will proceed. The driver merely needs to copy the packet from the ring buffer to the caller's buffer and return. File *ethread.c* contains the code:

```
/* ethread.c - ethread */

#include <xinu.h>

/*------------------------------------------------------------------------
 * ethread  -  Read an incoming packet on Intel Quark Ethernet
 *------------------------------------------------------------------------
 */
devcall ethread (
          struct dentry *devptr,        /* Entry in device switch table */
          char  *buf,                   /* Buffer for the packet        */
          int32 len                     /* Size of the buffer           */
        )
{
        struct  ethcblk *ethptr;        /* Ethertab entry pointer       */
        struct  eth_q_rx_desc *rdescptr;/* Pointer to the descriptor    */
        struct  netpacket *pktptr;      /* Pointer to packet            */
        uint32  framelen = 0;           /* Length of the incoming frame */
        bool8   valid_addr;
        int32   i;

        ethptr = &ethertab[devptr->dvminor];

        while(1) {

                /* Wait until there is a packet in the receive queue */

                wait(ethptr->isem);

                /* Point to the head of the descriptor list */

                rdescptr = (struct eth_q_rx_desc *)ethptr->rxRing +
                                                        ethptr->rxHead;
                pktptr = (struct netpacket*)rdescptr->buffer1;

                /* See if destination address is our unicast address */

                if(!memcmp(pktptr->net_ethdst, ethptr->devAddress, 6)) {
```

```
        valid_addr = TRUE;

/* See if destination address is the broadcast address */

} else if(!memcmp(pktptr->net_ethdst,
                    NetData.ethbcast,6)) {
        valid_addr = TRUE;

/* For multicast addresses, see if we should accept */

} else {
        valid_addr = FALSE;
        for(i = 0; i < (ethptr->ed_mcc); i++) {
                if(memcmp(pktptr->net_ethdst,
                        ethptr->ed_mca[i], 6) == 0){
                        valid_addr = TRUE;
                        break;
                }
        }
}

if(valid_addr == TRUE){ /* Accept this packet */

        /* Get the length of the frame */

        framelen = (rdescptr->status >> 16) & 0x00003FFF;

        /* Only return len characters to caller */

        if(framelen > len) {
                framelen = len;
        }

        /* Copy the packet into the caller's buffer */

        memcpy(buf, (void*)rdescptr->buffer1, framelen);
}

/* Increment the head of the descriptor list */

ethptr->rxHead += 1;
if(ethptr->rxHead >= ETH_QUARK_RX_RING_SIZE) {
        ethptr->rxHead = 0;
}
```

```
                /* Reset the descriptor to max possible frame len */

                rdescptr->buf1size = sizeof(struct netpacket);

                /* If we reach the end of the ring, mark the descriptor */

                if(ethptr->rxHead == 0) {
                        rdescptr->rdctl1 |= (ETH_QUARK_RDCTL1_RER);
                }

                /* Indicate that the descriptor is ready for DMA input */

                rdescptr->status = ETH_QUARK_RDST_OWN;

                if(valid_addr == TRUE) {
                        break;
                }
        }

        /* Return the number of bytes returned from the packet */

        return framelen;

}
```

Ethread consists of a loop that reads packets until it finds a packet that should be processed (i.e., valid). A packet is accepted if the packet is addressed to the computer's unicast address, the network broadcast address, or is one of the multicast addresses to which the computer is listening. Other packets are discarded.

Multicast. Like many hardware devices, the Galileo Ethernet does not recognize a large set of multicast addresses. Instead, the hardware receives a set of multicast packets, and relies on the software to decide whether a packet should be accepted. Our driver uses an array of multicast addresses, and checks each incoming multicast packet to see whether the address matches one of the addresses that an application has specified. An exercise explores the use of hardware filters.

To block until at least one packet is available, *ethread* waits on the input semaphore on each iteration of the loop. Once it proceeds beyond the call to *wait*, the code locates the next available ring descriptor and obtains a pointer to the buffer associated with the ring. It then examines the destination address on the packet in the buffer. When a valid packet is found, the code copies the packet to the caller's buffer, and returns the size of the packet to the caller. Whether a packet is accepted or skipped, *ethread* makes the descriptor available to the device (i.e., ready for the DMA engine to deposit another packet).

16.10 Writing To An Ethernet Device

Using DMA makes output as straightforward as input. An application calls *write* to send a packet, which invokes function *ethwrite*. As with input, the output side only interacts with the ring buffers: *ethwrite* copies the caller's buffer to the next available output buffer. File *ethwrite.c* contains the code:

```
/* ethwrite.c - ethwrite */

#include <xinu.h>

/*------------------------------------------------------------------------
 * ethwrite  -  enqueue packet for transmission on Intel Quark Ethernet
 *------------------------------------------------------------------------
 */
devcall ethwrite        (
          struct dentry *devptr,        /* Entry in device switch table */
          char  *buf,                   /* Buffer that hols a packet    */
          int32 len                     /* Length of the packet         */
        )
{
        struct  ethcblk *ethptr;        /* Pointer to control block     */
        struct  eth_q_csreg *csrptr;    /* Address of device CSRs       */
        volatile struct eth_q_tx_desc *descptr; /* Ptr to descriptor    */
        uint32 i;                       /* Counts bytes during copy     */

        ethptr = &ethertab[devptr->dvminor];

        csrptr = (struct eth_q_csreg *)ethptr->csr;

        /* Wait for an empty slot in the transmit descriptor ring */

        wait(ethptr->osem);

        /* Point to the tail of the descriptor ring */

        descptr = (struct eth_q_tx_desc *)ethptr->txRing + ethptr->txTail;

        /* Increment the tail index and wrap, if needed */

        ethptr->txTail += 1;
        if(ethptr->txTail >= ethptr->txRingSize) {
                ethptr->txTail = 0;
        }
```

```
        /* Add packet length to the descriptor */

        descptr->buf1size = len;

        /* Copy packet into the buffer associated with the descriptor   */

        for(i = 0; i < len; i++) {
                *((char *)descptr->buffer1 + i) = *((char *)buf + i);
        }

        /* Mark the descriptor if we are at the end of the ring */

        if(ethptr->txTail == 0) {
                descptr->ctrlstat = ETH_QUARK_TDCS_TER;
        } else {
                descptr->ctrlstat = 0;
        }

        /* Initialize the descriptor */

        descptr->ctrlstat |=
                (ETH_QUARK_TDCS_OWN | /* The desc is owned by DMA       */
                 ETH_QUARK_TDCS_IC  | /* Interrupt after transfer       */
                 ETH_QUARK_TDCS_LS  | /* Last segment of packet         */
                 ETH_QUARK_TDCS_FS);  /* First segment of packet        */

        /* Un-suspend DMA on the device */

        csrptr->tpdr = 1;

        return OK;
}
```

Ethwrite waits on the output semaphore, which blocks the calling process until an output ring descriptor is empty and available. The code then copies a packet from the caller's buffer into the packet buffer associated with the descriptor. If the device is currently idle, *ethwrite* must start the device. Starting the device is trivial: *ethwrite* assigns 1 to the *tpdr* register in the device's transmit control register. If the device is already running, the assignment has no effect; if the device is idle, the assignment starts the device, which will examine the next ring descriptor (the one into which the driver placed the packet to be transmitted).

16.11 Handling Interrupts From An Ethernet Device

One of the advantages of a DMA device arises because the DMA engine on the device handles many of the details. As a result, interrupt processing does not involve much interaction with the device. An interrupt occurs when an input or output operation completes successfully or when the DMA engine encounters an error. To determine the exact cause, the interrupt handler checks the *transmit interrupt* bit in the control register, and then checks the *receive interrupt* bit. Other interrupts are acknowledged (to reset the hardware), but otherwise ignored. File *ethhandler.c* contains the code:

```
/* ethhandler.c - ethhandler */

#include <xinu.h>

/*------------------------------------------------------------------------
 * ethhandler  -  Interrupt handler for Intel Quark Ethernet
 *------------------------------------------------------------------------
 */
interrupt       ethhandler(void)
{
        struct  ethcblk *ethptr;        /* Ethertab entry pointer       */
        struct  eth_q_csreg *csrptr;    /* Pointer to Ethernet CRSs     */
        struct  eth_q_tx_desc *tdescptr;/* Pointer to tx descriptor     */
        struct  eth_q_rx_desc *rdescptr;/* Pointer to rx descriptor     */
        volatile uint32 sr;             /* Copy of status register      */
        uint32  count;                  /* Variable used to count pkts  */

        ethptr = &ethertab[devtab[ETHER0].dvminor];

        csrptr = (struct eth_q_csreg *)ethptr->csr;

        /* Copy the status register into a local variable */

        sr = csrptr->sr;

        /* If there is no interrupt pending, return */

        if((csrptr->sr & ETH_QUARK_SR_NIS) == 0) {
                return;
        }

        /* Acknowledge the interrupt */

        csrptr->sr = sr;
```

```
/* Check status register to figure out the source of interrupt */

if (sr & ETH_QUARK_SR_TI) { /* Transmit interrupt */

        /* Pointer to the head of transmit desc ring */

        tdescptr = (struct eth_q_tx_desc *)ethptr->txRing +
                                                ethptr->txHead;

        /* Start packet count at zero */

        count = 0;

        /* Repeat until we process all the descriptor slots */

        while(ethptr->txHead != ethptr->txTail) {

                /* If the descriptor is owned by DMA, stop here */

                if(tdescptr->ctrlstat & ETH_QUARK_TDCS_OWN) {
                        break;
                }

                /* Descriptor was processed; increment count    */

                count++;

                /* Go to the next descriptor */

                tdescptr += 1;

                /* Increment the head of the transmit desc ring */

                ethptr->txHead += 1;
                if(ethptr->txHead >= ethptr->txRingSize) {
                        ethptr->txHead = 0;
                        tdescptr = (struct eth_q_tx_desc *)
                                                ethptr->txRing;
                }
        }

        /* 'count' packets were processed by DMA, and slots are */
        /* now free; signal the semaphore accordingly           */

        signaln(ethptr->osem, count);
```

```
        }
        if(sr & ETH_QUARK_SR_RI) { /* Receive interrupt */

                /* Get the pointer to the tail of the receive desc list */

                rdescptr = (struct eth_q_rx_desc *)ethptr->rxRing +
                                                        ethptr->rxTail;

                count = 0;      /* Start packet count at zero */

                /* Repeat until we have received                */
                /* maximum no. packets that can fit in queue    */

                while(count <= ethptr->rxRingSize) {

                        /* If the descriptor is owned by the DMA, stop */

                        if(rdescptr->status & ETH_QUARK_RDST_OWN) {
                                break;
                        }

                        /* Descriptor was processed; increment count    */
                        count++;

                        /* Go to the next descriptor */

                        rdescptr += 1;

                        /* Increment the tail index of the rx desc ring */

                        ethptr->rxTail += 1;
                        if(ethptr->rxTail >= ethptr->rxRingSize) {
                                ethptr->rxTail = 0;
                                rdescptr = (struct eth_q_rx_desc *)
                                                        ethptr->rxRing;
                        }
                }

                /* 'count' packets were received and are available,     */
                /*   so signal the semaphore accordingly                */

                signaln(ethptr->isem, count);
        }

        return;
}
```

Note that when an interrupt occurs, more than one packet may have been transmitted or received. The transmit interrupt code iterates through the descriptor ring, counts the available slots, and uses *signaln* to signal the output semaphore. The receive interrupt code iterates through the descriptor ring, counts available packets, and uses *signaln* to signal the input semaphore.

16.12 Ethernet Control Functions

The Ethernet driver supports three control functions: a caller can fetch the MAC address from the device, add a multicast address to the set of addresses that the driver accepts, or remove a multicast address from the set. The MAC address is trivial: because the MAC address is copied into the control block at startup, the code merely copies the address from the control block into the buffer the caller has provided (*arg1*).

Rather than incorporate code for multicast address manipulation into *ethcontrol*, the code has been placed into two functions. *Ethmcast_add* adds an address to the set (inserts it in the array that *ethread* examines), and *ethmcast_remove* deletes an address from the array.

```
/* ethcontrol.c - ethcontrol */

#include <xinu.h>

/*------------------------------------------------------------------------
 * ethcontrol - implement control function for a quark ethernet device
 *------------------------------------------------------------------------
 */
devcall ethcontrol (
        struct  dentry *devptr,         /* entry in device switch table */
        int32   func,                   /* control function             */
        int32   arg1,                   /* argument 1, if needed        */
        int32   arg2                    /* argument 2, if needed        */
        )
{
        struct  ethcblk *ethptr;        /* Ethertab entry pointer       */
        int32   retval = OK;            /* Return value of cntl function*/

        ethptr = &ethertab[devptr->dvminor];

        switch (func) {

                /* Get MAC address */

                case ETH_CTRL_GET_MAC:
```

```
                        memcpy((byte *)arg1, ethptr->devAddress,
                                        ETH_ADDR_LEN);
                        break;

                /* Add a multicast address */

                case ETH_CTRL_ADD_MCAST:
                        retval = ethmcast_add(ethptr, (byte *)arg1);
                        break;

                /* Remove a multicast address */

                case ETH_CTRL_REMOVE_MCAST:
                        retval = ethmcast_remove(ethptr, (byte *)arg1);
                        break;

                default:
                        return SYSERR;
        }

        return retval;
}
```

16.13 Perspective

DMA devices present an interesting irony to a programmer who must write a device driver. On the one hand, DMA hardware can be incredibly complex, and the documentation (which is called a *data sheet*) is often so difficult to understand that programmers find it impenetrable. Unlike a device with a few simple control and status registers, a DMA device requires a programmer to create complex data structures in memory and to communicate their location to the device. Furthermore, a programmer must understand exactly how and when the hardware sets response bits in the data structures and how the hardware interprets the requests that the operating system generates. On the other hand, once a programmer masters the documentation, the resulting driver code is usually smaller than the code for a non-DMA device. Thus, DMA devices have a steep learning curve, but offer the reward of both higher performance and smaller driver code.

16.14 Summary

A device that uses Direct Memory Access (DMA) can move an arbitrary block of data between the device and memory without using the processor to fetch individual words of data. A DMA device typically uses a descriptor ring in memory, where each node in the ring points to one buffer. Once the driver points the hardware to a node of the ring, the DMA engine performs the operation and moves to the next node on the ring automatically.

The chief advantage of a DMA device lies in lower overhead: the device only needs to interrupt the processor once per block instead of once per byte or once per word. The driver code for a DMA device is simpler than the code for a conventional device because the driver does not need to perform low-level operations.

EXERCISES

16.1 Read about Ethernet packets and find the minimum packet size. At 100 Mbps, how many packets can arrive per second?

16.2 Our code merely accepts all multicast packets and allows the driver to filter. Consult the hardware data sheet, and rewrite the code to use the multicast filtering that the hardware provides.

16.3 In the previous exercise, is it necessary for the driver to check incoming packets even if hardware filtering is enabled? Explain.

16.4 Build a test program that transmits Ethernet packets as fast as possible. How many large packets can you send per second? How many small packets?

16.5 The current driver is complex and the code is somewhat difficult to read. Can you rewrite the code to allocate ring descriptor and buffers statically? Why or why not?

Chapter Contents

17.1 Introduction, 357
17.2 Required Functionality, 358
17.3 Simultaneous Conversations, Timeouts, And Processes, 359
17.4 A Consequence Of The Design, 359
17.5 ARP Functions, 360
17.6 Definition Of A Network Packet, 371
17.7 The Network Input Process, 373
17.8 Definitions For IP, 377
17.9 IP Functions, 377
17.10 Definition Of The UDP Table, 388
17.11 UDP Functions, 389
17.12 Internet Control Message Protocol, 403
17.13 Dynamic Host Configuration Protocol, 404
17.14 Perspective, 412
17.15 Summary, 413

17

A Minimal Internet Protocol Stack

The lure of the distant and the difficult is deceptive.
The great opportunity is where you are.

— John Burroughs

17.1 Introduction

Most embedded systems are connected — they use computer networks to access remote services and storage facilities. As a result, protocol software has become a required part of even small embedded operating systems. The previous chapter describes a basic Ethernet device driver that can send and receive packets. Although an Ethernet device can transfer packets across a single network, additional communication software is required to permit applications to communicate across the Internet. In particular, the *TCP/Internet Protocol Suite* defines the protocols used for Internet communication. The protocols are organized into conceptual layers, and an implementation is known as a *protocol stack*.

A complete TCP/IP stack contains many protocols, and requires much more than a single chapter to describe. Therefore, this chapter describes a minimal implementation that is sufficiently powerful to support the remote disk and remote file systems covered later in the book. It provides a brief description without delving into the details of the protocols; the reader is referred to other texts from the author that explain the protocol suite and a full implementation.

17.2 Required Functionality

Our implementation of Internet protocols allows a process running on Xinu to communicate with an application running on a remote computer in the Internet (e.g., a PC, Mac, or Unix system, such as Linux or Solaris). It is possible to identify a remote computer and exchange messages with the computer. The system includes a timeout mechanism that allows a receiver to be informed if no message is received within a specified timeout.

In terms of protocols, our implementation supports basic Internet protocols:

- IP Internet Protocol†
- UDP User Datagram Protocol
- ARP Address Resolution Protocol
- DHCP Dynamic Host Configuration Protocol
- ICMP Internet Control Message Protocol

The *Internet Protocol (IP)* defines the format of an internet packet, which is known as a *datagram*. Each datagram is carried in the data area of an Ethernet frame. The Internet Protocol also defines the address format. Our implementation does not support IPv4 options or features such as fragmentation (i.e., it is not a complete implementation). Packet forwarding follows the pattern used in most end systems: our IPv4 software knows the computer's IP address, address mask for the local network, and a single *default router* address; if a destination is not on the local network, the packet is sent to the default router.

The *User Datagram Protocol (UDP)* defines a set of 16-bit *port numbers* that an operating system uses to identify a specific application program. Communicating applications must agree on the port numbers they will use. Port numbers allow simultaneous communication without interference: an application can interact with one remote server while a second application interacts with another. Our software allows a process to specify a port number at runtime.

The *Address Resolution Protocol (ARP)* provides two functions. Before another computer can send IP packets to our system, the computer must send an ARP packet that requests our Ethernet address and our system must respond with an ARP reply. Similarly, before our system can send IP packets to another computer, it first sends an ARP request to obtain the computer's Ethernet address, then uses the Ethernet address to send IP packets.

The *Dynamic Host Configuration Protocol (DHCP)* provides a mechanism that a computer can use to obtain an IP address, an address mask for the network, and the IP address of a *default router*. The computer broadcasts a request, and a DHCP server running on the network sends a response. Usually, DHCP is invoked at startup because the information must be obtained before normal Internet communication is possible. Our implementation does not invoke DHCP immediately at startup. Instead, it waits until a process attempts to obtain a local IP address.

---

†Throughout this text, *IP* will refer to version 4 of the Internet Protocol, which is sometimes written *IPv4* to distinguish it from the successor, *IPv6*.

The *Internet Control Message Protocol (ICMP)* provides error and informational messages that support IP. Our implementation only handles the two ICMP messages used by the *ping* program: *Echo Request* and *Echo Reply*. Because the code for ICMP is large, we describe the structure of the protocol software without showing all the details; the code is available on the web site for the text.†

17.3 Simultaneous Conversations, Timeouts, And Processes

How should protocol software be organized? How many processes are needed? There is no easy answer. Our minimal stack implements the set of protocols described above with an elegant design that has a single network input process, named *netin*,‡ and a single IP output process named *ipout*. The software uses *recvtime* to handle *timeout-and-retransmission*. That is, after transmitting a message, a sender calls *recvtime* to wait for a response. When a response arrives, the *netin* process calls a function that sends a message to the waiting process, and *recvtime* returns the message. If the timer expires, *recvtime* returns value *TIMEOUT*. The protocol software provides coordination between application processes and *netin*. We will see how the software handles timeout. Figure 17.1 illustrates the basic idea: the *netin* process places an incoming UDP packet in a queue associated with a UDP port number, and uses an incoming ARP packet to supply information for an ARP table entry. In either case, if an application process is waiting for the incoming packet, *netin* sends a message to allow the waiting process to run.

Figure 17.1 The operation of the *netin* process that handles incoming packets by using UDP and ARP tables to coordinate with processes that are waiting for the packets.

17.4 A Consequence Of The Design

The use of a *netin* process has an important consequence for the overall design of protocol software. To understand the consequence, one must know three facts. First, *netin* is the only process that reads and handles incoming packets — if *netin* blocks, no more incoming packets will be handled. Second, incoming IP packets sometimes cause

†URL: *xinu.cs.purdue.edu*

‡The code for *netin* can be found later in the chapter in file *net.c* on page 374.

the transmission of a reply. For example, the *ping* protocol, which is used to test whether a computer is reachable, requires the transmission of a response whenever a *ping* request arrives. Third, transmission of an outgoing IP packet may require a preliminary ARP exchange. That is, before it can send an IP packet, the protocol software may need to send an ARP request and receive an ARP reply.

Taken together, the three facts mean that *netin* must never attempt to transmit an IP packet, or a deadlock can occur. To summarize:

> *Because the* netin *process must remain running to receive incoming packets,* netin *must never execute code that blocks waiting for a packet to arrive. In particular,* netin *cannot execute code that sends an IP packet, such as a* ping *reply.*

To decouple IP reply output from input, our implementation uses an extra process, *ipout*. When a local application sends a packet, transmission can proceed directly. When protocol software replies to an incoming packet (e.g., a response to a *ping* request), however, the *netin* process enqueues the IP packet for the IP output process to send. Figure 17.2 illustrates the decoupling by showing how a *ping* request and reply pass through the two processes.

Figure 17.2 The process structure with *ipout* decoupled from *netin*, and the example of a *ping* request that causes a reply to be sent.

17.5 ARP Functions

Before two computers on an Ethernet can communicate using the Internet Protocol, they must learn each other's Ethernet addresses.† The protocol exchanges two messages: computer A broadcasts an *ARP request* that contains an IP address. Whichever computer on the network has the IP address in the request sends an *ARP response* that specifies its Ethernet address. When a response arrives, an entry is added to a table that is known as an *ARP cache*. The entry contains the remote computer's IP address and its Ethernet address. Subsequent transmissions to the same destination extract the information from the ARP cache without sending another request.

Our implementation stores ARP information in array *arpcache*. Struct *arpentry* defines the contents of each entry in the array to consist of: a state field (which specifies whether the entry is currently unused, being filled in, or already filled in), an IP address,

†Technically, they learn *MAC* addresses, but in our examples, MAC addresses will be Ethernet addresses.

the corresponding Ethernet address, and a process ID. If the entry is in the pending state, the process ID field contains the ID of the process that is waiting for the information to arrive. File *arp.h* defines the ARP packet format (when used on an Ethernet) and the format of an ARP cache entry.

```
/* arp.h */

/* Items related to ARP - definition of cache and the packet format */

#define ARP_HALEN       6               /* Size of Ethernet MAC address */
#define ARP_PALEN       4               /* Size of IP address           */

#define ARP_HTYPE       1               /* Ethernet hardware type       */
#define ARP_PTYPE       0x0800          /* IP protocol type             */

#define ARP_OP_REQ      1               /* Request op code              */
#define ARP_OP_RPLY     2               /* Reply op code                */

#define ARP_SIZ         16              /* Number of entries in a cache */

#define ARP_RETRY       3               /* Num. retries for ARP request */

#define ARP_TIMEOUT     300             /* Retry timer in milliseconds  */

/* State of an ARP cache entry */

#define AR_FREE         0               /* Slot is unused               */
#define AR_PENDING      1               /* Resolution in progress       */
#define AR_RESOLVED     2               /* Entry is valid               */

#pragma pack(2)
struct  arppacket {                     /* ARP packet for IP & Ethernet */
        byte    arp_ethdst[ETH_ADDR_LEN];/* Ethernet dest. MAC addr     */
        byte    arp_ethsrc[ETH_ADDR_LEN];/* Ethernet source MAC address */
        uint16  arp_ethtype;            /* Ethernet type field          */
        uint16  arp_htype;              /* ARP hardware type            */
        uint16  arp_ptype;              /* ARP protocol type            */
        byte    arp_hlen;               /* ARP hardware address length  */
        byte    arp_plen;               /* ARP protocol address length  */
        uint16  arp_op;                 /* ARP operation                */
        byte    arp_sndha[ARP_HALEN];   /* ARP sender's Ethernet addr   */
        uint32  arp_sndpa;              /* ARP sender's IP address      */
        byte    arp_tarha[ARP_HALEN];   /* ARP target's Ethernet addr   */
        uint32  arp_tarpa;              /* ARP target's IP address      */
};
```

```
#pragma pack()

struct  arpentry {                      /* Entry in the ARP cache       */
        int32   arstate;                /* State of the entry           */
        uint32  arpaddr;                /* IP address of the entry      */
        pid32   arpid;                  /* Waiting process or -1        */
        byte    arhaddr[ARP_HALEN];     /* Ethernet address of the entry*/
};

extern struct   arpentry arpcache[];
```

ARP uses the same packet format for both requests and responses; a field in the header specifies the type as a request or response. In each case, the packet contains the sender's IP address and Ethernet address as well as the target's IP address and Ethernet address. In a request, the target's Ethernet address is unknown, so the field contains zeroes.

Our ARP software consists of four functions, *arp_init*, *arp_resolve*, *arp_in*, and *arp_alloc*. All four functions reside in a single source file, *arp.c*:

```
/* arp.c - arp_init, arp_resolve, arp_in, arp_alloc, arp_ntoh, arp_hton */

#include <xinu.h>

struct  arpentry  arpcache[ARP_SIZ];    /* ARP cache                    */

/*------------------------------------------------------------------------
 * arp_init  -  Initialize ARP cache for an Ethernet interface
 *------------------------------------------------------------------------
 */
void    arp_init(void)
{
        int32   i;                      /* ARP cache index              */

        for (i=1; i<ARP_SIZ; i++) {     /* Initialize cache to empty    */
                arpcache[i].arstate = AR_FREE;
        }
}

/*------------------------------------------------------------------------
 * arp_resolve  -  Use ARP to resolve an IP address to an Ethernet address
 *------------------------------------------------------------------------
 */
status  arp_resolve (
         uint32 nxthop,                 /* Next-hop address to resolve  */
```

```
 byte   mac[ETH_ADDR_LEN]       /* Array into which Ethernet    */
)                               /*   address should be placed   */
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  arppacket apkt;         /* Local packet buffer          */
        int32   i;                      /* Index into arpcache          */
        int32   slot;                   /* ARP table slot to use        */
        struct  arpentry  *arptr;       /* Ptr to ARP cache entry       */
        int32   msg;                    /* Message returned by recvtime */

        /* Use MAC broadcast address for IP limited broadcast */

        if (nxthop == IP_BCAST) {
                memcpy(mac, NetData.ethbcast, ETH_ADDR_LEN);
                return OK;
        }

        /* Use MAC broadcast address for IP network broadcast */

        if (nxthop == NetData.ipbcast) {
                memcpy(mac, NetData.ethbcast, ETH_ADDR_LEN);
                return OK;
        }

        /* Ensure only one process uses ARP at a time */

        mask = disable();

        /* See if next hop address is already present in ARP cache */

        for (i=0; i<ARP_SIZ; i++) {
                arptr = &arpcache[i];
                if (arptr->arstate == AR_FREE) {
                        continue;
                }
                if (arptr->arpaddr == nxthop) { /* Adddress is in cache */
                        break;
                }
        }

        if (i < ARP_SIZ) {      /* Entry was found */

                /* If entry is resolved - handle and return */

                if (arptr->arstate == AR_RESOLVED) {
```

```
                memcpy(mac, arptr->arhaddr, ARP_HALEN);
                restore(mask);
                return OK;
        }

        /* Entry is already pending -  return error because     */
        /*      only one process can be waiting at a time       */

        if (arptr->arstate == AR_PENDING) {
                restore(mask);
                return SYSERR;
        }
}

/* IP address not in cache -  allocate a new cache entry and    */
/*      send an ARP request to obtain the answer                */

slot = arp_alloc();
if (slot == SYSERR) {
        restore(mask);
        return SYSERR;
}

arptr = &arpcache[slot];
arptr->arstate = AR_PENDING;
arptr->arpaddr = nxthop;
arptr->arpid = currpid;

/* Hand-craft an ARP Request packet */

memcpy(apkt.arp_ethdst, NetData.ethbcast, ETH_ADDR_LEN);
memcpy(apkt.arp_ethsrc, NetData.ethucast, ETH_ADDR_LEN);
apkt.arp_ethtype = ETH_ARP;       /* Packet type is ARP         */
apkt.arp_htype = ARP_HTYPE;       /* Hardware type is Ethernet  */
apkt.arp_ptype = ARP_PTYPE;       /* Protocol type is IP        */
apkt.arp_hlen = 0xff & ARP_HALEN; /* Ethernet MAC size in bytes */
apkt.arp_plen = 0xff & ARP_PALEN; /* IP address size in bytes   */
apkt.arp_op = 0xffff & ARP_OP_REQ;/* ARP type is Request        */
memcpy(apkt.arp_sndha, NetData.ethucast, ARP_HALEN);
apkt.arp_sndpa = NetData.ipucast; /* IP address of interface    */
memset(apkt.arp_tarha, '\0', ARP_HALEN); /* Target HA is unknown*/
apkt.arp_tarpa = nxthop;          /* Target protocol address    */

/* Convert ARP packet from host to net byte order */
```

```
        arp_hton(&apkt);

        /* Convert Ethernet header from host to net byte order */

        eth_hton((struct netpacket *)&apkt);

        /* Send the packet ARP_RETRY times and await response */

        msg = recvclr();
        for (i=0; i<ARP_RETRY; i++) {
                write(ETHER0, (char *)&apkt, sizeof(struct arppacket));
                msg = recvtime(ARP_TIMEOUT);
                if (msg == TIMEOUT) {
                        continue;
                } else if (msg == SYSERR) {
                        restore(mask);
                        return SYSERR;
                } else {        /* entry is resolved */
                        break;
                }
        }

        /* If no response, return TIMEOUT */

        if (msg == TIMEOUT) {
                arptr->arstate = AR_FREE;   /* Invalidate cache entry */
                restore(mask);
                return TIMEOUT;
        }

        /* Return hardware address */

        memcpy(mac, arptr->arhaddr, ARP_HALEN);
        restore(mask);
        return OK;
}


/*------------------------------------------------------------------------
 * arp_in  -  Handle an incoming ARP packet
 *------------------------------------------------------------------------
 */
void    arp_in (
          struct arppacket *pktptr      /* Ptr to incoming packet       */
        )
```

```
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  arppacket apkt;         /* Local packet buffer          */
        int32   slot;                   /* Slot in cache                */
        struct  arpentry  *arptr;       /* Ptr to ARP cache entry       */
        bool8   found;                  /* Is the sender's address in   */
                                        /*   the cache?                 */

        /* Convert packet from network order to host order */

        arp_ntoh(pktptr);

        /* Verify ARP is for IPv4 and Ethernet */

        if ( (pktptr->arp_htype != ARP_HTYPE) ||
             (pktptr->arp_ptype != ARP_PTYPE) ) {
                freebuf((char *)pktptr);
                return;
        }

        /* Ensure only one process uses ARP at a time */

        mask = disable();

        /* Search cache for sender's IP address */

        found = FALSE;

        for (slot=0; slot < ARP_SIZ; slot++) {
                arptr = &arpcache[slot];

                /* Skip table entries that are unused */

                if (arptr->arstate == AR_FREE) {
                        continue;
                }

                /* If sender's address matches, we've found it */

                if (arptr->arpaddr == pktptr->arp_sndpa) {
                        found = TRUE;
                        break;
                }
        }
```

```
        if (found) {

                /* Update sender's hardware address */

                memcpy(arptr->arhaddr, pktptr->arp_sndha, ARP_HALEN);

                /* If a process was waiting, inform the process */

                if (arptr->arstate == AR_PENDING) {
                        /* Mark resolved and notify waiting process */
                        arptr->arstate = AR_RESOLVED;
                        send(arptr->arpid, OK);
                }
        }

        /* For an ARP reply, processing is complete */

        if (pktptr->arp_op == ARP_OP_RPLY) {
                freebuf((char *)pktptr);
                restore(mask);
                return;
        }

        /* The following is for an ARP request packet: if the local     */
        /*  machine is not the target or the local IP address is not    */
        /*  yet known, ignore the request (i.e., processing is complete)*/

        if ((!NetData.ipvalid) ||
                        (pktptr->arp_tarpa != NetData.ipucast)) {
                freebuf((char *)pktptr);
                restore(mask);
                return;
        }

        /* Request has been sent to the local machine's address.  So,   */
        /*   add sender's info to cache, if not already present         */

        if (!found) {
                slot = arp_alloc();
                if (slot == SYSERR) {   /* Cache is full */
                        kprintf("ARP cache overflow on interface\n");
                        freebuf((char *)pktptr);
                        restore(mask);
                        return;
                }
```

```
                arptr = &arpcache[slot];
                arptr->arpaddr = pktptr->arp_sndpa;
                memcpy(arptr->arhaddr, pktptr->arp_sndha, ARP_HALEN);
                arptr->arstate = AR_RESOLVED;
        }

        /* Hand-craft an ARP reply packet and send back to requester    */

        memcpy(apkt.arp_ethdst, pktptr->arp_sndha, ARP_HALEN);
        memcpy(apkt.arp_ethsrc, NetData.ethucast, ARP_HALEN);
        apkt.arp_ethtype= ETH_ARP;              /* Frame carries ARP    */
        apkt.arp_htype  = ARP_HTYPE;            /* Hardware is Ethernet */
        apkt.arp_ptype  = ARP_PTYPE;            /* Protocol is IP       */
        apkt.arp_hlen   = ARP_HALEN;            /* Ethernet address size*/
        apkt.arp_plen   = ARP_PALEN;            /* IP address size      */
        apkt.arp_op     = ARP_OP_RPLY;          /* Type is Reply        */

        /* Insert local Ethernet and IP address in sender fields        */

        memcpy(apkt.arp_sndha, NetData.ethucast, ARP_HALEN);
        apkt.arp_sndpa = NetData.ipucast;

        /* Copy target Ethernet and IP addresses from request packet */

        memcpy(apkt.arp_tarha, pktptr->arp_sndha, ARP_HALEN);
        apkt.arp_tarpa = pktptr->arp_sndpa;

        /* Convert ARP packet from host to network byte order */

        arp_hton(&apkt);

        /* Convert the Ethernet header to network byte order */

        eth_hton((struct netpacket *)&apkt);

        /* Send the reply */

        write(ETHER0, (char *)&apkt, sizeof(struct arppacket));
        freebuf((char *)pktptr);
        restore(mask);
        return;
}

/*------------------------------------------------------------------------
 * arp_alloc  -  Find a free slot or kick out an entry to create one
```

```
 *------------------------------------------------------------------------
 */
int32   arp_alloc ()
{
        int32   slot;                   /* Slot in ARP cache            */

        /* Search for a free slot */

        for (slot=0; slot < ARP_SIZ; slot++) {
                if (arpcache[slot].arstate == AR_FREE) {
                        memset((char *)&arpcache[slot],
                                        NULLCH, sizeof(struct arpentry));
                        return slot;
                }
        }

        /* Search for a resolved entry */

        for (slot=0; slot < ARP_SIZ; slot++) {
                if (arpcache[slot].arstate == AR_RESOLVED) {
                        memset((char *)&arpcache[slot],
                                        NULLCH, sizeof(struct arpentry));
                        return slot;
                }
        }

        /* At this point, all slots are pending (should not happen) */

        kprintf("ARP cache size exceeded\n");

        return SYSERR;
}

/*------------------------------------------------------------------------
 * arp_ntoh  -  Convert ARP packet fields from net to host byte order
 *------------------------------------------------------------------------
 */
void    arp_ntoh(
        struct arppacket *pktptr
        )
{
        pktptr->arp_htype = ntohs(pktptr->arp_htype);
        pktptr->arp_ptype = ntohs(pktptr->arp_ptype);
        pktptr->arp_op    = ntohs(pktptr->arp_op);
        pktptr->arp_sndpa = ntohl(pktptr->arp_sndpa);
```

```
        pktptr->arp_tarpa = ntohl(pktptr->arp_tarpa);
}

/*------------------------------------------------------------------------
 * arp_hton  -  Convert ARP packet fields from net to host byte order
 *------------------------------------------------------------------------
 */
void    arp_hton(
        struct arppacket *pktptr
        )
{
        pktptr->arp_htype = htons(pktptr->arp_htype);
        pktptr->arp_ptype = htons(pktptr->arp_ptype);
        pktptr->arp_op    = htons(pktptr->arp_op);
        pktptr->arp_sndpa = htonl(pktptr->arp_sndpa);
        pktptr->arp_tarpa = htonl(pktptr->arp_tarpa);
}
```

Function *arp_init* is called once when the system starts. It marks each entry in the ARP cache free, and creates a mutual exclusion semaphore that ensures only one process will attempt to change the ARP cache (e.g., insert an entry) at any time. Functions *arp_resolve* and *arp_in* are used to handle address lookup for outgoing IP packets and to process incoming ARP packets, respectively. The final function, *arp_alloc*, is called to allocate an entry in the table whenever a new item must be added.

Function *arp_resolve* is called when an IP packet is ready to be sent. *Arp_resolve* takes two arguments: the first specifies the IP address of a computer for which an Ethernet address is needed; the second is a pointer to an array that will hold the Ethernet address.

Although the code may seem complex, there are only three cases: the IP address is a broadcast address, the information is already in the ARP cache, or the information is not known. For an IP broadcast address, *arp_resolve* copies the Ethernet broadcast address into the array specified by the second argument. If the information is present in the cache, *arp_resolve* finds the correct entry, copies the Ethernet address from the entry into the caller's array, and returns to the caller without sending any packets over the network.

In the case where the requested mapping is not in the cache, *arp_resolve* must send packets over the network to obtain the information. The exchange involves sending a request and waiting for a reply. *Arp_resolve* creates an entry in the table, marks the entry *AR_PENDING*, forms an ARP request packet, broadcasts the packet on the local network, and then waits for a reply. As discussed above, *arp_resolve* uses *recvtime* to wait. The call to *recvtime* will return if a response arrives or the timer expires, whichever occurs first. In the next section, we will describe how an incoming packet is processed and how a message is sent to a waiting process.

The code is more complex than we have described because *arp_resolve* does not merely give up if a timeout occurs. Instead, our implementation is designed to retry the operation: it sends a request and waits for a reply *ARP_RETRY* times before it returns *TIMEOUT* to the caller.

Arp_in, the second major ARP function, runs when an incoming ARP packet arrives. The *netin* process examines the type field in each incoming Ethernet packet. If it finds the ARP packet type (*0x806*), *netin* calls function *arp_in* to handle the packet. *Arp_in* must handle two cases: either the packet is a request that was initiated by another computer or it is a reply, possibly to a request that we have sent.

The protocol specifies that when either type of packet arrives, ARP must examine the sender's information (IP address and Ethernet address), and update the local cache accordingly. If a process is waiting for the reply, *arp_in* sends a message to the process.

Because an ARP request is broadcast, all computers on the network receive each request. Therefore, after it updates the sender's information, *arp_in* checks the target IP address in a request to determine whether the request is for the local system or some other computer on the network. If the request is for another computer, *arp_in* returns without taking further action. If the target IP address in the incoming request matches the IP address of the local system, *arp_in* sends an ARP reply. *Arp_in* forms a reply in variable *apkt*. Once all fields of the packet have been filled in, the code calls *write* on the Ethernet device to transmit the reply back to the requester.

17.6 Definition Of A Network Packet

Our minimal implementation of network protocols combines IP, UDP, ICMP, and Ethernet. That is, we use a single data structure, named *netpacket*, to describe an Ethernet packet that carries an IP datagram which either carries a UDP message or an ICMP message. File *net.h* defines *netpacket* as well as constants, such as the Ethernet type values used for ARP and IP. Finally, the file defines a *network* data structure that holds address information for the local machine (both Ethernet and IP addresses), and variable *NetData* to be a struct of type *network*.

```
/* net.h */

#define NETSTK          8192            /* Stack size for network setup */
#define NETPRIO         500             /* Network startup priority     */
#define NETBOOTFILE     128             /* Size of the netboot filename */

/* Constants used in the networking code */

#define ETH_ARP     0x0806              /* Ethernet type for ARP        */
#define ETH_IP      0x0800              /* Ethernet type for IP         */
#define ETH_IPv6    0x86DD              /* Ethernet type for IPv6       */

/* Format of an Ethernet packet carrying IPv4 and UDP */

#pragma pack(2)
struct  netpacket       {
        byte    net_ethdst[ETH_ADDR_LEN];/* Ethernet dest. MAC address  */
        byte    net_ethsrc[ETH_ADDR_LEN];/* Ethernet source MAC address */
        uint16  net_ethtype;            /* Ethernet type field          */
        byte    net_ipvh;               /* IP version and hdr length    */
        byte    net_iptos;              /* IP type of service           */
        uint16  net_iplen;              /* IP total packet length       */
        uint16  net_ipid;               /* IP datagram ID               */
        uint16  net_ipfrag;             /* IP flags & fragment offset   */
        byte    net_ipttl;              /* IP time-to-live              */
        byte    net_ipproto;            /* IP protocol (actually type)  */
        uint16  net_ipcksum;            /* IP checksum                  */
        uint32  net_ipsrc;              /* IP source address            */
        uint32  net_ipdst;              /* IP destination address       */
        union {
         struct {
          uint16        net_udpsport;   /* UDP source protocol port     */
          uint16        net_udpdport;   /* UDP destination protocol port*/
          uint16        net_udplen;     /* UDP total length             */
          uint16        net_udpcksum;   /* UDP checksum                 */
          byte          net_udpdata[1500-28];/* UDP payload (1500-above)*/
         };
         struct {
          byte          net_ictype;     /* ICMP message type            */
          byte          net_iccode;     /* ICMP code field (0 for ping) */
          uint16        net_iccksum;    /* ICMP message checksum        */
          uint16        net_icident;    /* ICMP identifier              */
          uint16        net_icseq;      /* ICMP sequence number         */
          byte          net_icdata[1500-28];/* ICMP payload (1500-above)*/
         };
        };
};
#pragma pack()
```

```
#define PACKLEN sizeof(struct netpacket)

extern  bpid32  netbufpool;             /* ID of net packet buffer pool */

struct  network {
        uint32  ipucast;
        uint32  ipbcast;
        uint32  ipmask;
        uint32  ipprefix;
        uint32  iprouter;
        uint32  bootserver;
        bool8   ipvalid;
        byte    ethucast[ETH_ADDR_LEN];
        byte    ethbcast[ETH_ADDR_LEN];
        char    bootfile[NETBOOTFILE];
};

extern  struct  network NetData;        /* Local Network Interface      */
```

17.7 The Network Input Process

At startup, Xinu calls function *net_init* to initialize data structures and start the network processes. After it creates the network buffer pool and initializes global variables, *net_init* calls *arp_init*, *udp_init*, and *icmp_init*. It then initializes the IP output queue to empty, and creates the *netin* and *netout* processes.

The *netin* process repeatedly allocates a buffer, waits for the next packet, and then performs *packet demultiplexing*. That is, *netin* uses information in each incoming packet to decide which protocol to use to process the packet. When it reads a packet from an Ethernet, *netin* uses the Ethernet *type field* in the packet to determine whether the Ethernet packet is carrying an ARP message or an IP datagram. In the case of ARP, *netin* passes the packet to *arp_in*, which performs input processing on arriving ARP packets. In the case of an IP datagram, *netin* passes the packet to *ip_in*, which performs input processing on arriving IP datagrams.

The code contains two additional cases: one for *IPv6* and a default for all other packets. In each case, the action is the same: *netin* drops (i.e., discards) the packet without further processing. To do so, *netin* calls *freebuf* and passes a pointer to the buffer. The case for *IPv6* has been separated from the default to indicate where one would add code for IPv6.

File *net.c* contains the code for both *net_init* and *netin*.

```
/* net.c - net_init, netin, eth_hton */

#include <xinu.h>
#include <stdio.h>

struct  network NetData;
bpid32  netbufpool;

/*------------------------------------------------------------------------
 * net_init  -  Initialize network data structures and processes
 *------------------------------------------------------------------------
 */

void    net_init (void)
{
        int32   nbufs;                  /* Total no of buffers          */

        /* Initialize the network data structure */

        memset((char *)&NetData, NULLCH, sizeof(struct network));

        /* Obtain the Ethernet MAC address */

        control(ETHER0, ETH_CTRL_GET_MAC, (int32)NetData.ethucast, 0);

        memset((char *)NetData.ethbcast, 0xFF, ETH_ADDR_LEN);

        /* Create the network buffer pool */

        nbufs = UDP_SLOTS * UDP_QSIZ + ICMP_SLOTS * ICMP_QSIZ + 1;

        netbufpool = mkbufpool(PACKLEN, nbufs);

        /* Initialize the ARP cache */

        arp_init();

        /* Initialize UDP */

        udp_init();

        /* Initialize ICMP */

        icmp_init();

        /* Initialize the IP output queue */
```

```
        ipoqueue.iqhead = 0;
        ipoqueue.iqtail = 0;
        ipoqueue.iqsem = semcreate(0);
        if((int32)ipoqueue.iqsem == SYSERR) {
                panic("Cannot create ip output queue semaphore");
                return;
        }

        /* Create the IP output process */

        resume(create(ipout, NETSTK, NETPRIO, "ipout", 0, NULL));

        /* Create a network input process */

        resume(create(netin, NETSTK, NETPRIO, "netin", 0, NULL));
}


/*------------------------------------------------------------------------
 * netin  -  Repeatedly read and process the next incoming packet
 *------------------------------------------------------------------------
 */

process netin ()
{
        struct  netpacket *pkt;         /* Ptr to current packet        */
        int32   retval;                 /* Return value from read       */

        /* Do forever: read a packet from the network and process */

        while(1) {

                /* Allocate a buffer */

                pkt = (struct netpacket *)getbuf(netbufpool);

                /* Obtain next packet that arrives */

                retval = read(ETHER0, (char *)pkt, PACKLEN);
                if(retval == SYSERR) {
                        panic("Cannot read from Ethernet\n");
                }

                /* Convert Ethernet Type to host order */

                eth_ntoh(pkt);
```

```
                /* Demultiplex on Ethernet type */

                switch (pkt->net_ethtype) {

                    case ETH_ARP:                       /* Handle ARP   */
                        arp_in((struct arppacket *)pkt);
                        continue;

                    case ETH_IP:                        /* Handle IP    */
                        ip_in(pkt);
                        continue;

                    case ETH_IPv6:                      /* Handle IPv6  */
                        freebuf((char *)pkt);
                        continue;

                    default:    /* Ignore all other incoming packets    */
                        freebuf((char *)pkt);
                        continue;
                }
        }
}

/*------------------------------------------------------------------------
 * eth_hton  -  Convert Ethernet type field to network byte order
 *------------------------------------------------------------------------
 */
void    eth_hton(
          struct netpacket *pktptr
        )
{
        pktptr->net_ethtype = htons(pktptr->net_ethtype);
}


/*------------------------------------------------------------------------
 * eth_ntoh  -  Convert Ethernet type field to host byte order
 *------------------------------------------------------------------------
 */
void    eth_ntoh(
          struct netpacket *pktptr
        )
{
        pktptr->net_ethtype = ntohs(pktptr->net_ethtype);
}
```

17.8 Definitions For IP

File *ip.h* contains definitions needed for functions that handle the Internet Protocol (IP), including constants for the local broadcast address, type values for UDP and ICMP, and header constants. The file also defines struct *iqentry* that specifies the contents of an entry in the IP output queue.

```
/* ip.h  -  Constants related to Internet Protocol version 4 (IPv4) */

#define IP_BCAST        0xffffffff      /* IP local broadcast address   */
#define IP_THIS         0xffffffff      /* "this host" src IP address   */
#define IP_ALLZEROS     0x00000000      /* The all-zeros IP address     */

#define IP_ICMP         1               /* ICMP protocol type for IP    */
#define IP_UDP          17              /* UDP protocol type for IP     */

#define IP_ASIZE        4               /* Bytes in an IP address       */
#define IP_HDR_LEN      20              /* Bytes in an IP header        */
#define IP_VH           0x45            /* IP version and hdr length    */

#define IP_OQSIZ        8               /* Size of IP output queue      */

/* Queue of outgoing IP packets waiting for ipout process */

struct  iqentry {
        int32   iqhead;                 /* Index of next packet to send */
        int32   iqtail;                 /* Index of next free slot      */
        sid32   iqsem;                  /* Semaphore that counts pkts   */
        struct  netpacket *iqbuf[IP_OQSIZ];/* Circular packet queue     */
};

extern  struct  iqentry ipoqueue;       /* Network output queue         */
```

17.9 IP Functions

Our IP software uses eight functions that are specified in file *ip.c*: *ip_in*, *ip_send*, *ip_local*, *ip_out*, *ipcksum*, *ip_hton*, *ip_ntoh*, and *ip_enqueue*. In addition, the file contains the code for *ipout*, the IP output process.

On input, *ip_in* passes valid datagrams to *ip_local*, which examines the type field in the datagram. Datagrams carrying UDP are passed to *udp_in*, datagrams carrying *ICMP* are passed to *icmp_in*, and other datagrams are dropped.

```
/* ip.c - ip_in, ip_send, ip_local, ip_out, ipcksum, ip_hton, ip_ntoh,  */
/*               ipout, ip_enqueue                                      */

#include <xinu.h>

struct  iqentry ipoqueue;               /* Queue of outgoing packets    */

/*------------------------------------------------------------------------
 * ip_in  -  Handle an IP packet that has arrived over a network
 *------------------------------------------------------------------------
 */

void    ip_in(
          struct netpacket *pktptr      /* Pointer to the packet        */
        )
{
        int32   icmplen;                /* Length of ICMP message       */

        /* Verify checksum */

        if (ipcksum(pktptr) != 0) {
                kprintf("IP header checksum failed\n\r");
                freebuf((char *)pktptr);
                return;
        }

        /* Convert IP header fields to host order */

        ip_ntoh(pktptr);

        /* Ensure version and length are valid */

        if (pktptr->net_ipvh != 0x45) {
                kprintf("IP version failed\n\r");
                freebuf((char *)pktptr);
                return;
        }

        /* Verify encapsulated prototcol checksums and then convert     */
        /*      the encapsulated headers to host byte order             */

        switch (pktptr->net_ipproto) {

            case IP_UDP:
                /* Skipping UDP checksum for now */
```

```
            udp_ntoh(pktptr);
            break;

        case IP_ICMP:
            icmplen = pktptr->net_iplen - IP_HDR_LEN;
            if (icmp_cksum((char *)&pktptr->net_ictype,icmplen) != 0){
                    freebuf((char *)pktptr);
                    return;
            }
            icmp_ntoh(pktptr);
            break;

        default:
            break;
    }

    /* Deliver 255.255.255.255 to local stack */

    if (pktptr->net_ipdst == IP_BCAST) {
            ip_local(pktptr);
            return;
    }

    /* If we do not yet have a valid address, accept UDP packets    */
    /*      (to get DHCP replies) and drop others                   */

    if (!NetData.ipvalid) {
            if (pktptr->net_ipproto == IP_UDP) {
                    ip_local(pktptr);
                    return;
            } else {
                    freebuf((char *)pktptr);
                    return;
            }
    }

    /* If packet is destined for us, accept it; otherwise, drop it  */

    if ( (pktptr->net_ipdst == NetData.ipucast) ||
         (pktptr->net_ipdst == NetData.ipbcast) ||
         (pktptr->net_ipdst == IP_BCAST)  ) {
            ip_local(pktptr);
            return;
```

```
        } else {

                /* Drop the packet */
                freebuf((char *)pktptr);
                return;
        }
}


/*------------------------------------------------------------------------
 * ip_send  -  Send an outgoing IP datagram from the local stack
 *------------------------------------------------------------------------
 */

status  ip_send(
          struct netpacket *pktptr      /* Pointer to the packet        */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        uint32  dest;                   /* Destination of the datagram  */
        int32   retval;                 /* Return value from functions  */
        uint32  nxthop;                 /* Next-hop address             */

        mask = disable();

        /* Pick up the IP destination address from the packet */

        dest = pktptr->net_ipdst;

        /* Loop back to local stack if destination 127.0.0.0/8 */

        if ((dest&0xff000000) == 0x7f000000) {
                ip_local(pktptr);
                restore(mask);
                return OK;
        }

        /* Loop back if the destination matches our IP unicast address  */

        if (dest == NetData.ipucast) {
                ip_local(pktptr);
                restore(mask);
                return OK;
        }
```

```
        /* Broadcast if destination is 255.255.255.255 */

        if ( (dest == IP_BCAST) ||
             (dest == NetData.ipbcast) ) {
                memcpy(pktptr->net_ethdst, NetData.ethbcast,
                                                        ETH_ADDR_LEN);
                retval = ip_out(pktptr);
                restore(mask);
                return retval;
        }

        /* If destination is on the local network, next hop is the      */
        /*      destination; otherwise, next hop is default router      */


        if ( (dest & NetData.ipmask) == NetData.ipprefix) {

                /* Next hop is the destination itself */
                nxthop = dest;

        } else {

                /* Next hop is default router on the network */
                nxthop = NetData.iprouter;

        }

        if (nxthop == 0) {      /* Dest. invalid or no default route    */
                freebuf((char *)pktptr);
                return SYSERR;
        }

        /* Resolve the next-hop address to get a MAC address */

        retval = arp_resolve(nxthop, pktptr->net_ethdst);
        if (retval != OK) {
                freebuf((char *)pktptr);
                return SYSERR;
        }

        /* Send the packet */

        retval = ip_out(pktptr);
        restore(mask);
        return retval;
}
```

```
/*------------------------------------------------------------------------
 * ip_local  -  Deliver an IP datagram to the local stack
 *------------------------------------------------------------------------
 */
void    ip_local(
          struct netpacket *pktptr      /* Pointer to the packet        */
        )
{
        /* Use datagram contents to determine how to process */

        switch (pktptr->net_ipproto) {

            case IP_UDP:
                udp_in(pktptr);
                return;

            case IP_ICMP:
                icmp_in(pktptr);
                return;

            default:
                freebuf((char *)pktptr);
                return;
        }
}


/*------------------------------------------------------------------------
 *  ip_out  -  Transmit an outgoing IP datagram
 *------------------------------------------------------------------------
 */
status  ip_out(
          struct netpacket *pktptr      /* Pointer to the packet        */
        )
{
        uint16  cksum;                  /* Checksum in host byte order  */
        int32   len;                    /* Length of ICMP message       */
        int32   pktlen;                 /* Length of entire packet      */
        int32   retval;                 /* Value returned by write      */

        /* Compute total packet length */

        pktlen = pktptr->net_iplen + ETH_HDR_LEN;
```

```
/* Convert encapsulated protocol to network byte order */

switch (pktptr->net_ipproto) {

    case IP_UDP:

                pktptr->net_udpcksum = 0;
                udp_hton(pktptr);

                /* ...skipping UDP checksum computation */

                break;

    case IP_ICMP:
                icmp_hton(pktptr);

                /* Compute ICMP checksum */

                pktptr->net_iccksum = 0;
                len = pktptr->net_iplen-IP_HDR_LEN;
                cksum = icmp_cksum((char *)&pktptr->net_ictype,
                                                    len);
                pktptr->net_iccksum = 0xffff & htons(cksum);
                break;

    default:
                break;
}

/* Convert IP fields to network byte order */

ip_hton(pktptr);

/* Compute IP header checksum */

pktptr->net_ipcksum = 0;
cksum = ipcksum(pktptr);
pktptr->net_ipcksum = 0xffff & htons(cksum);

/* Convert Ethernet fields to network byte order */

eth_hton(pktptr);
```

```
        /* Send packet over the Ethernet */

        retval = write(ETHER0, (char*)pktptr, pktlen);
        freebuf((char *)pktptr);

        if (retval == SYSERR) {
                return SYSERR;
        } else {
                return OK;
        }
}

/*------------------------------------------------------------------------
 * ipcksum  -  Compute the IP header checksum for a datagram
 *------------------------------------------------------------------------
 */

uint16  ipcksum(
         struct  netpacket *pkt         /* Pointer to the packet        */
        )
{
        uint16  *hptr;                  /* Ptr to 16-bit header values  */
        int32   i;                      /* Counts 16-bit values in hdr  */
        uint16  word;                   /* One 16-bit word              */
        uint32  cksum;                  /* Computed value of checksum   */

        hptr= (uint16 *) &pkt->net_ipvh;

        /* Sum 16-bit words in the packet */

        cksum = 0;
        for (i=0; i<10; i++) {
                word = *hptr++;
                cksum += (uint32) htons(word);
        }

        /* Add in carry, and take the ones-complement */

        cksum += (cksum >> 16);
        cksum = 0xffff & ~cksum;
```

```
        /* Use all-1s for zero */

        if (cksum == 0xffff) {
                cksum = 0;
        }
        return (uint16) (0xffff & cksum);
}


/*------------------------------------------------------------------------
 * ip_ntoh  -  Convert IP header fields to host byte order
 *------------------------------------------------------------------------
 */
void    ip_ntoh(
          struct netpacket *pktptr
        )
{
        pktptr->net_iplen = ntohs(pktptr->net_iplen);
        pktptr->net_ipid = ntohs(pktptr->net_ipid);
        pktptr->net_ipfrag = ntohs(pktptr->net_ipfrag);
        pktptr->net_ipsrc = ntohl(pktptr->net_ipsrc);
        pktptr->net_ipdst = ntohl(pktptr->net_ipdst);
}

/*------------------------------------------------------------------------
 * ip_hton  -  Convert IP header fields to network byte order
 *------------------------------------------------------------------------
 */
void    ip_hton(
          struct netpacket *pktptr

        )
{
        pktptr->net_iplen = htons(pktptr->net_iplen);
        pktptr->net_ipid = htons(pktptr->net_ipid);
        pktptr->net_ipfrag = htons(pktptr->net_ipfrag);
        pktptr->net_ipsrc = htonl(pktptr->net_ipsrc);
        pktptr->net_ipdst = htonl(pktptr->net_ipdst);
}


/*------------------------------------------------------------------------
 *  ipout  -  Process that transmits IP packets from the IP output queue
 *------------------------------------------------------------------------
 */
```

```
process ipout(void)
{
        struct  netpacket *pktptr;      /* Pointer to next the packet   */
        struct  iqentry   *ipqptr;      /* Pointer to IP output queue   */
        uint32  destip;                 /* Destination IP address       */
        uint32  nxthop;                 /* Next hop IP address          */
        int32   retval;                 /* Value returned by functions  */

        ipqptr = &ipoqueue;

        while(1) {

                /* Obtain next packet from the IP output queue */

                wait(ipqptr->iqsem);
                pktptr = ipqptr->iqbuf[ipqptr->iqhead++];
                if (ipqptr->iqhead >= IP_OQSIZ) {
                        ipqptr->iqhead= 0;
                }

                /* Fill in the MAC source address */

                memcpy(pktptr->net_ethsrc, NetData.ethucast, ETH_ADDR_LEN);

                /* Extract destination address from packet */

                destip = pktptr->net_ipdst;


                /* Sanity check: packets sent to ioout should *not*     */
                /*      contain a broadcast address.                    */

                if ((destip == IP_BCAST)||(destip == NetData.ipbcast)) {
                        kprintf("ipout: encountered a broadcast\n");
                        freebuf((char *)pktptr);
                        continue;
                }

                /* Check whether destination is the local computer */

                if (destip == NetData.ipucast) {
                        ip_local(pktptr);
                        continue;
                }
```

```
                /* Check whether destination is on the local net */

                if ( (destip & NetData.ipmask) == NetData.ipprefix) {

                        /* Next hop is the destination itself */

                        nxthop = destip;
                } else {

                        /* Next hop is default router on the network */

                        nxthop = NetData.iprouter;
                }

                if (nxthop == 0) {  /* Dest. invalid or no default route*/
                        freebuf((char *)pktptr);
                        continue;
                }

                /* Use ARP to resolve next-hop address */

                retval = arp_resolve(nxthop, pktptr->net_ethdst);
                if (retval != OK) {
                        freebuf((char *)pktptr);
                        continue;
                }

                /* Use ipout to Convert byte order and send */

                ip_out(pktptr);
        }
}


/*------------------------------------------------------------------------
 *  ip_enqueue  -  Deposit an outgoing IP datagram on the IP output queue
 *------------------------------------------------------------------------
 */
status  ip_enqueue(
          struct netpacket *pktptr      /* Pointer to the packet        */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  iqentry *iptr;          /* Ptr. to network output queue */
```

```
        /* Ensure only one process accesses output queue at a time */

        mask = disable();

        /* Enqueue packet on network output queue */

        iptr = &ipoqueue;
        if (semcount(iptr->iqsem) >= IP_OQSIZ) {
                kprintf("ipout: output queue overflow\n");
                freebuf((char *)pktptr);
                restore(mask);
                return SYSERR;
        }
        iptr->iqbuf[iptr->iqtail++] = pktptr;
        if (iptr->iqtail >= IP_OQSIZ) {
                iptr->iqtail = 0;
        }
        signal(iptr->iqsem);
        restore(mask);
        return OK;
}
```

When a local application sends a datagram, *ip_send* is invoked. When the *netin* process needs to send a reply, it calls *ip_enqueue* to enqueue the packet for the IP output process to handle.

17.10 Definition Of The UDP Table

UDP maintains a table that specifies the pairs of UDP endpoints that are currently in use. A single endpoint consists of an IP address and a UDP port number. Therefore, an entry in the table has four fields that specify two endpoints: one for a remote computer and one for the local computer.

To act as a server that can receive a packet from an arbitrary remote computer, a process allocates a table entry, fills in the local endpoint information, and leaves the remote endpoint unspecified. To act as a client that communicates with a specific remote computer, a process allocates a table entry and fills in both the local and remote endpoint information.

In addition to endpoint information, each entry in the UDP table contains a queue of packets that have arrived from the remote system (i.e., packets where the endpoints specified in the packet match those in the table entry). An entry in the UDP table is described by struct *udpentry*; file *udp.h* defines the structure as well as associated symbolic constants.

```
/* udp.h - Declarations pertaining to User Datagram Protocol (UDP) */

#define UDP_SLOTS       6               /* Number of open UDP endpoints */
#define UDP_QSIZ        8               /* Packets enqueued per endpoint*/

#define UDP_DHCP_CPORT  68              /* Port number for DHCP client  */
#define UDP_DHCP_SPORT  67              /* Port number for DHCP server  */

/* Constants for the state of an entry */

#define UDP_FREE        0               /* Entry is unused              */
#define UDP_USED        1               /* Entry is being used          */
#define UDP_RECV        2               /* Entry has a process waiting  */

#define UDP_ANYIF       -2              /* Register an endpoint for any */
                                        /*   interface on the machine   */

#define UDP_HDR_LEN     8               /* Bytes in a UDP header        */

struct  udpentry {                      /* Entry in the UDP endpoint tbl*/
        int32   udstate;                /* State of entry: free/used    */
        uint32  udremip;                /* Remote IP address (zero      */
                                        /*   means "don't care")        */
        uint16  udremport;              /* Remote protocol port number  */
        uint16  udlocport;              /* Local protocol port number   */
        int32   udhead;                 /* Index of next packet to read */
        int32   udtail;                 /* Index of next slot to insert */
        int32   udcount;                /* Count of packets enqueued    */
        pid32   udpid;                  /* ID of waiting process        */
        struct  netpacket *udqueue[UDP_QSIZ];/* Circular packet queue   */
};

extern  struct  udpentry udptab[];
```

17.11 UDP Functions

To permit applications to communicate over the Internet, the UDP interface is designed to allow applications to send and receive UDP messages and to act as either a client or a server. Our UDP software includes eight interface functions†: *udp_init*, *udp_in*, *udp_register*, *udp_recv*, *udp_recvaddr*, *udp_send*, *udp_sendto*, and *udp_release*. The functions are collected into a single file, *udp.c*. Following the file, the text describes each UDP function.

†The code also contains two internal functions, *udp_ntoh* and *udp_hton*, that translate between host byte order and network byte order.

```
/* udp.c - udp_init, udp_in, udp_register, udp_send, udp_sendto,        */
/*              udp_recv, udp_recvaddr, udp_release, udp_ntoh, udp_hton */

#include <xinu.h>

struct  udpentry udptab[UDP_SLOTS];     /* Table of UDP endpoints       */

/*------------------------------------------------------------------------
 * udp_init  -  Initialize all entries in the UDP endpoint table
 *------------------------------------------------------------------------
 */
void    udp_init(void)
{

        int32   i;                      /* Index into the UDP table */

        for(i=0; i<UDP_SLOTS; i++) {
                udptab[i].udstate = UDP_FREE;
        }

        return;
}


/*------------------------------------------------------------------------
 * udp_in  -  Handle an incoming UDP packet
 *------------------------------------------------------------------------
 */
void    udp_in(
          struct netpacket *pktptr      /* Pointer to the packet        */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        int32   i;                      /* Index into udptab            */
        struct  udpentry *udptr;        /* Pointer to a udptab entry    */


        /* Ensure only one process can access the UDP table at a time   */

        mask = disable();

        for (i=0; i<UDP_SLOTS; i++) {
            udptr = &udptab[i];
            if (udptr->udstate == UDP_FREE) {
                        continue;
```

```
            }

            if ((pktptr->net_udpdport == udptr->udlocport)  &&
                    ((udptr->udremport == 0) ||
                        (pktptr->net_udpsport == udptr->udremport)) &&
                 (  ((udptr->udremip==0)     ||
                        (pktptr->net_ipsrc == udptr->udremip)))    ) {

                /* Entry matches incoming packet */

                if (udptr->udcount < UDP_QSIZ) {
                        udptr->udcount++;
                        udptr->udqueue[udptr->udtail++] = pktptr;
                        if (udptr->udtail >= UDP_QSIZ) {
                                udptr->udtail = 0;
                        }
                        if (udptr->udstate == UDP_RECV) {
                                udptr->udstate = UDP_USED;
                                send (udptr->udpid, OK);
                        }
                        restore(mask);
                        return;
                }
            }
        }

        /* No match - simply discard packet */

        freebuf((char *) pktptr);
        restore(mask);
        return;
}

/*------------------------------------------------------------------------
 * udp_register  -  Register a remote IP, remote port & local port to
 *                    receive incoming UDP messages from the specified
 *                    remote site sent to the specified local port
 *------------------------------------------------------------------------
 */
uid32   udp_register (
         uint32 remip,                  /* Remote IP address or zero    */
         uint16 remport,                /* Remote UDP protocol port     */
         uint16 locport                 /* Local UDP protocol port      */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
```

```
int32   slot;                   /* Index into udptab            */
struct  udpentry *udptr;        /* Pointer to udptab entry      */

/* Ensure only one process can access the UDP table at a time   */

mask = disable();

/* See if request already registered */

for (slot=0; slot<UDP_SLOTS; slot++) {
        udptr = &udptab[slot];
        if (udptr->udstate == UDP_FREE) {
                continue;
        }

        /* Look at this entry in table */

        if ( (remport == udptr->udremport) &&
             (locport == udptr->udlocport) &&
             (remip   == udptr->udremip  ) ) {

                /* Request is already in the table */

                restore(mask);
                return SYSERR;
        }
}

/* Find a free slot and allocate it */

for (slot=0; slot<UDP_SLOTS; slot++) {
        udptr = &udptab[slot];
        if (udptr->udstate != UDP_FREE) {
                continue;
        }
        udptr->udlocport = locport;
        udptr->udremport = remport;
        udptr->udremip = remip;
        udptr->udcount = 0;
        udptr->udhead = udptr->udtail = 0;
        udptr->udpid = -1;
        udptr->udstate = UDP_USED;
        restore(mask);
        return slot;
}
```

```
        restore(mask);
        return SYSERR;
}

/*------------------------------------------------------------------------
 * udp_recv  -  Receive a UDP packet
 *------------------------------------------------------------------------
 */
int32   udp_recv (
         uid32  slot,                   /* Slot in table to use         */
         char   *buff,                  /* Buffer to hold UDP data      */
         int32  len,                    /* Length of buffer             */
         uint32 timeout                 /* Read timeout in msec         */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  udpentry *udptr;        /* Pointer to udptab entry      */
        umsg32  msg;                    /* Message from recvtime()      */
        struct  netpacket *pkt;         /* Pointer to packet being read */
        int32   i;                      /* Counts bytes copied          */
        int32   msglen;                 /* Length of UDP data in packet */
        char    *udataptr;              /* Pointer to UDP data          */

        /* Ensure only one process can access the UDP table at a time   */

        mask = disable();

        /* Verify that the slot is valid */

        if ((slot < 0) || (slot >= UDP_SLOTS)) {
                restore(mask);
                return SYSERR;
        }

        /* Get pointer to table entry */

        udptr = &udptab[slot];

        /* Verify that the slot has been registered and is valid */

        if (udptr->udstate != UDP_USED) {
                restore(mask);
                return SYSERR;
        }
```

```
        /* Wait for a packet to arrive */

        if (udptr->udcount == 0) {              /* No packet is waiting */
                udptr->udstate = UDP_RECV;
                udptr->udpid = currpid;
                msg = recvclr();
                msg = recvtime(timeout);        /* Wait for a packet    */
                udptr->udstate = UDP_USED;
                if (msg == TIMEOUT) {
                        restore(mask);
                        return TIMEOUT;
                } else if (msg != OK) {
                        restore(mask);
                        return SYSERR;
                }
        }

        /* Packet has arrived -- dequeue it */

        pkt = udptr->udqueue[udptr->udhead++];
        if (udptr->udhead >= UDP_QSIZ) {
                udptr->udhead = 0;
        }
        udptr->udcount--;

        /* Copy UDP data from packet into caller's buffer */

        msglen = pkt->net_udplen - UDP_HDR_LEN;
        udataptr = (char *)pkt->net_udpdata;
        if (len < msglen) {
                msglen = len;
        }
        for (i=0; i<msglen; i++) {
                *buff++ = *udataptr++;
        }
        freebuf((char *)pkt);
        restore(mask);
        return msglen;
}

/*------------------------------------------------------------------------
 * udp_recvaddr  -  Receive a UDP packet and record the sender's address
 *------------------------------------------------------------------------
 */
int32   udp_recvaddr (
```

```
         uid32  slot,                  /* Slot in table to use         */
         uint32 *remip,                /* Loc for remote IP address    */
         uint16 *remport,              /* Loc for remote protocol port */
         char   *buff,                 /* Buffer to hold UDP data      */
         int32  len,                   /* Length of buffer             */
         uint32 timeout                /* Read timeout in msec         */
        )
{
        intmask mask;                  /* Saved interrupt mask         */
        struct  udpentry *udptr;       /* Pointer to udptab entry      */
        umsg32  msg;                   /* Message from recvtime()      */
        struct  netpacket *pkt;        /* Pointer to packet being read */
        int32   msglen;                /* Length of UDP data in packet */
        int32   i;                     /* Counts bytes copied          */
        char    *udataptr;             /* Pointer to UDP data          */

        /* Ensure only one process can access the UDP table at a time   */

        mask = disable();

        /* Verify that the slot is valid */

        if ((slot < 0) || (slot >= UDP_SLOTS)) {
                restore(mask);
                return SYSERR;
        }

        /* Get pointer to table entry */

        udptr = &udptab[slot];

        /* Verify that the slot has been registered and is valid */

        if (udptr->udstate != UDP_USED) {
                restore(mask);
                return SYSERR;
        }

        /* Wait for a packet to arrive */

        if (udptr->udcount == 0) {             /* No packet is waiting */
                udptr->udstate = UDP_RECV;
                udptr->udpid = currpid;
                msg = recvclr();
                msg = recvtime(timeout);       /* Wait for a packet    */
```

```
                udptr->udstate = UDP_USED;
                if (msg == TIMEOUT) {
                        restore(mask);
                        return TIMEOUT;
                } else if (msg != OK) {
                        restore(mask);
                        return SYSERR;
                }
        }

        /* Packet has arrived -- dequeue it */

        pkt = udptr->udqueue[udptr->udhead++];
        if (udptr->udhead >= UDP_QSIZ) {
                udptr->udhead = 0;
        }

        /* Record sender's IP address and UDP port number */

        *remip = pkt->net_ipsrc;
        *remport = pkt->net_udpsport;

        udptr->udcount--;

        /* Copy UDP data from packet into caller's buffer */

        msglen = pkt->net_udplen - UDP_HDR_LEN;
        udataptr = (char *)pkt->net_udpdata;
        if (len < msglen) {
                msglen = len;
        }
        for (i=0; i<msglen; i++) {
                *buff++ = *udataptr++;
        }
        freebuf((char *)pkt);
        restore(mask);
        return msglen;
}

/*------------------------------------------------------------------------
 * udp_send  -  Send a UDP packet using info in a UDP table entry
 *------------------------------------------------------------------------
 */
status  udp_send (
         uid32  slot,                   /* Table slot to use            */
```

```
         char   *buff,                  /* Buffer of UDP data           */
         int32  len                     /* Length of data in buffer     */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  netpacket *pkt;         /* Pointer to packet buffer     */
        int32   pktlen;                 /* Total packet length          */
        static  uint16 ident = 1;       /* Datagram IDENT field         */
        char    *udataptr;              /* Pointer to UDP data          */
        uint32  remip;                  /* Remote IP address to use     */
        uint16  remport;                /* Remote protocol port to use  */
        uint16  locport;                /* Local protocol port to use   */
        uint32  locip;                  /* Local IP address taken from  */
                                        /*   the interface              */
        struct  udpentry *udptr;        /* Pointer to table entry       */

        /* Ensure only one process can access the UDP table at a time   */

        mask = disable();

        /* Verify that the slot is valid */

        if ( (slot < 0) || (slot >= UDP_SLOTS) ) {
                restore(mask);
                return SYSERR;
        }

        /* Get pointer to table entry */

        udptr = &udptab[slot];

        /* Verify that the slot has been registered and is valid */

        if (udptr->udstate == UDP_FREE) {
                restore(mask);
                return SYSERR;
        }

        /* Verify that the slot has a specified remote address */

        remip = udptr->udremip;
        if (remip == 0) {
                restore(mask);
                return SYSERR;
        }
```

```
locip = NetData.ipucast;
remport = udptr->udremport;
locport = udptr->udlocport;

/* Allocate a network buffer to hold the packet */

pkt = (struct netpacket *)getbuf(netbufpool);

if ((int32)pkt == SYSERR) {
        restore(mask);
        return SYSERR;
}

/* Compute packet length as UDP data size + fixed header size   */

pktlen = ((char *)&pkt->net_udpdata - (char *)pkt) + len;

/* Create a UDP packet in pkt */

memcpy((char *)pkt->net_ethsrc,NetData.ethucast,ETH_ADDR_LEN);
pkt->net_ethtype = 0x0800;      /* Type is IP                   */
pkt->net_ipvh = 0x45;           /* IP version and hdr length    */
pkt->net_iptos = 0x00;          /* Type of service              */
pkt->net_iplen= pktlen - ETH_HDR_LEN;/* Total IP datagram length*/
pkt->net_ipid = ident++;        /* Datagram gets next IDENT     */
pkt->net_ipfrag = 0x0000;       /* IP flags & fragment offset   */
pkt->net_ipttl = 0xff;          /* IP time-to-live              */
pkt->net_ipproto = IP_UDP;      /* Datagram carries UDP         */
pkt->net_ipcksum = 0x0000;      /* initial checksum             */
pkt->net_ipsrc = locip;         /* IP source address            */
pkt->net_ipdst = remip;         /* IP destination address       */

pkt->net_udpsport = locport;    /* Local UDP protocol port      */
pkt->net_udpdport = remport;    /* Remote UDP protocol port     */
pkt->net_udplen = (uint16)(UDP_HDR_LEN+len); /* UDP length      */
pkt->net_udpcksum = 0x0000;     /* Ignore UDP checksum          */
udataptr = (char *) pkt->net_udpdata;
for (; len>0; len--) {
        *udataptr++ = *buff++;
}

/* Call ipsend to send the datagram */

ip_send(pkt);
restore(mask);
```

```
        return OK;
}


/*------------------------------------------------------------------------
 * udp_sendto  -  Send a UDP packet to a specified destination
 *------------------------------------------------------------------------
 */
status  udp_sendto (
         uid32  slot,                   /* UDP table slot to use        */
         uint32 remip,                  /* Remote IP address to use     */
         uint16 remport,                /* Remote protocol port to use  */
         char   *buff,                  /* Buffer of UDP data           */
         int32  len                     /* Length of data in buffer     */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  netpacket *pkt;         /* Pointer to a packet buffer   */
        int32   pktlen;                 /* Total packet length          */
        static  uint16 ident = 1;       /* Datagram IDENT field         */
        struct  udpentry *udptr;        /* Pointer to a UDP table entry */
        char    *udataptr;              /* Pointer to UDP data          */

        /* Ensure only one process can access the UDP table at a time   */

        mask = disable();

        /* Verify that the slot is valid */

        if ( (slot < 0) || (slot >= UDP_SLOTS) ) {
                restore(mask);
                return SYSERR;
        }

        /* Get pointer to table entry */

        udptr = &udptab[slot];

        /* Verify that the slot has been registered and is valid */

        if (udptr->udstate == UDP_FREE) {
                restore(mask);
                return SYSERR;
        }
```

```
        /* Allocate a network buffer to hold the packet */

        pkt = (struct netpacket *)getbuf(netbufpool);

        if ((int32)pkt == SYSERR) {
                restore(mask);
                return SYSERR;
        }

        /* Compute packet length as UDP data size + fixed header size   */

        pktlen = ((char *)&pkt->net_udpdata - (char *)pkt) + len;

        /* Create UDP packet in pkt */

        memcpy((char *)pkt->net_ethsrc,NetData.ethucast,ETH_ADDR_LEN);
        pkt->net_ethtype = 0x0800;      /* Type is IP */
        pkt->net_ipvh = 0x45;           /* IP version and hdr length    */
        pkt->net_iptos = 0x00;          /* Type of service              */
        pkt->net_iplen= pktlen - ETH_HDR_LEN;/* total IP datagram length*/
        pkt->net_ipid = ident++;        /* Datagram gets next IDENT     */
        pkt->net_ipfrag = 0x0000;       /* IP flags & fragment offset   */
        pkt->net_ipttl = 0xff;          /* IP time-to-live              */
        pkt->net_ipproto = IP_UDP;      /* Datagram carries UDP         */
        pkt->net_ipcksum = 0x0000;      /* Initial checksum             */
        pkt->net_ipsrc = NetData.ipucast;/* IP source address           */
        pkt->net_ipdst = remip;         /* IP destination address       */
        pkt->net_udpsport = udptr->udlocport;/* local UDP protocol port */
        pkt->net_udpdport = remport;    /* Remote UDP protocol port     */
        pkt->net_udplen = (uint16)(UDP_HDR_LEN+len); /* UDP length      */
        pkt->net_udpcksum = 0x0000;     /* Ignore UDP checksum          */
        udataptr = (char *) pkt->net_udpdata;
        for (; len>0; len--) {
                *udataptr++ = *buff++;
        }

        /* Call ipsend to send the datagram */

        ip_send(pkt);
        restore(mask);
        return OK;
}


/*------------------------------------------------------------------------
```

```
 * udp_release  -  Release a previously-registered UDP slot
 *------------------------------------------------------------------------
 */
status  udp_release (
         uid32  slot                    /* Table slot to release        */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  udpentry *udptr;        /* Pointer to udptab entry      */
        struct  netpacket *pkt;         /* pointer to packet being read */

        /* Ensure only one process can access the UDP table at a time   */

        mask = disable();

        /* Verify that the slot is valid */

        if ( (slot < 0) || (slot >= UDP_SLOTS) ) {
                restore(mask);
                return SYSERR;
        }

        /* Get pointer to table entry */

        udptr = &udptab[slot];

        /* Verify that the slot has been registered and is valid */

        if (udptr->udstate == UDP_FREE) {
                restore(mask);
                return SYSERR;
        }

        /* Defer rescheduling to prevent freebuf from switching context */

        resched_cntl(DEFER_START);
        while (udptr->udcount > 0) {
                pkt = udptr->udqueue[udptr->udhead++];
                if (udptr->udhead >= UDP_QSIZ) {
                        udptr->udhead = 0;
                }
                freebuf((char *)pkt);
                udptr->udcount--;
        }
        udptr->udstate = UDP_FREE;
```

```
        resched_cntl(DEFER_STOP);
        restore(mask);
        return OK;
}

/*------------------------------------------------------------------------
 * udp_ntoh  -  Convert UDP header fields from net to host byte order
 *------------------------------------------------------------------------
 */
void    udp_ntoh(
          struct netpacket *pktptr
        )
{
        pktptr->net_udpsport = ntohs(pktptr->net_udpsport);
        pktptr->net_udpdport = ntohs(pktptr->net_udpdport);
        pktptr->net_udplen = ntohs(pktptr->net_udplen);
        return;
}

/*------------------------------------------------------------------------
 * udp_hton  -  Convert packet header fields from host to net byte order
 *------------------------------------------------------------------------
 */
void    udp_hton(
          struct netpacket *pktptr
        )
{
        pktptr->net_udpsport = htons(pktptr->net_udpsport);
        pktptr->net_udpdport = htons(pktptr->net_udpdport);
        pktptr->net_udplen = htons(pktptr->net_udplen);
        return;
}
```

Udp_init. The initialization function is easiest to understand. The system calls *udp_init* once during startup, and *udp_init* sets the state of each entry in the UDP table to indicate that the entry is unused.

Udp_in. The *ip_in* process calls function *udp_in* when a packet arrives carrying a UDP message. Argument *pktptr* points to the incoming packet. *Udp_in* searches the UDP table to see if an entry in the table matches the port numbers and IP addresses in the current packet. If there is no match, the incoming packet is dropped — *udp_in* calls *freebuf* to return the buffer to the buffer pool. If a match does occur, *udp_in* inserts the incoming packet in the queue associated with the table entry. If the queue is full, *udp_in* drops the packet. When it inserts a packet in the queue, *udp_in* checks to see if a process is waiting for a packet to arrive (state *UDP_RECV*), and sends a message to

the waiting process. Note that at any time, only one process can be waiting for an entry in the table; if multiple processes need to use an entry to communicate, they must coordinate among themselves.

Udp_register. Before it can use UDP to communicate, an application must call *udp_register* to specify that it expects to receive incoming packets sent to a specific protocol port. The application can act as a client by specifying a remote IP address or can act as a server to accept packets from an arbitrary sender. *Udp_register* allocates an entry in the UDP table, records the remote and local protocol port and IP address information in the entry, and creates a queue to hold incoming packets.

Udp_recv. Once a local port number has been registered, an application can call *udp_recv* to extract a packet from a table entry. Arguments to the call specify a slot in the UDP table, a buffer to hold the incoming message, and a buffer length. The slot must have been registered previously using *udp_register*. In terms of coordination, *udp_recv* uses the same paradigm as ARP. If no packet is waiting (i.e., the queue for the entry is empty), *udp_recv* blocks and waits for the amount of time specified by the last argument. When a UDP packet arrives, *netin* calls *udp_in*. The code in *udp_in* finds the appropriate entry in the UDP table, and if an application process is waiting, sends a message to the waiting process. Thus, if a packet arrives within the specified time, *udp_recv* copies the UDP data to the caller's buffer and returns the length. If the timer expires before a packet arrives, *udp_recv* returns *TIMEOUT*.

Udp_recvaddr. When it acts as a server, a process must learn the address of the client that contacted it. A server process calls *udp_recvaddr*, which acts like *udp_recv* except that the call returns both an incoming packet and the address of the sender. The server can use the address to send a reply.

Udp_send. A process calls *udp_send* to transmit a UDP message. Arguments specify a slot in the UDP table, the address of a message in memory, and the length of the message. The slot must have been registered with *udp_register*. *Udp_send* creates an Ethernet packet that contains an IP datagram carrying the specified UDP message. *Udp_send* obtains the IP address and port numbers from the table entry.

Udp_release. Once a process has finished using a UDP endpoint, the process can call *udp_release* to release the table entry. If packets are enqueued in an entry, *udp_release* returns each to the buffer pool before marking the table entry free.

17.12 Internet Control Message Protocol

Our implementation of ICMP only handles the two message types used by the *ping* program: *ICMP Echo Request* and *ICMP Echo Reply*. Despite the restriction on message types, the code contains seven major functions: *icmp_init*, *icmp_in*, *icmp_out*, *icmp_register*, *icmp_send*, *icmp_recv*, and *icmp_release*.

As with other parts of the protocol stack, the network input function initializes ICMP by calling *icmp_init*. The network input process calls *icmp_in* when an ICMP packet arrives. An application process calls *icmp_register* to register its use of a remote IP address, then uses *icmp_send* to send a *ping* request and *icmp_recv* to receive a re-

ply. Finally, once it has finished, the application calls *icmp_release* to release the remote IP address and allow other processes to use it.

Although it is not shown in the text,† the functions follow the same general structure as the UDP functions. A trick is used to associate *ping* replies with requests: the identification field in an outgoing *ping* packet is the slot in the *ping* table (i.e., the index of an entry). When a reply arrives, the reply contains the same identification, which *icmp_in* uses as an index into the array. Thus, unlike UDP, the ICMP code never needs to search the table. Of course, the identification field alone is not sufficient: once a table entry has been identified, *icmp_in* verifies that the IP source address in the reply matches the IP address in the entry. Recall from the discussion earlier in the chapter that the *netin* process is running when *icmp_in* is called. Therefore, if a reply is needed, *icmp_in* does not send the reply directly. Instead, it calls *ip_enqueue* to enqueue the outgoing packet for the IP output process to send.

17.13 Dynamic Host Configuration Protocol

When it boots, a computer must obtain its IP address, the IP address of a default router, and the address mask used on the local network. The protocol used to obtain information at startup is known as the *Dynamic Host Configuration Protocol* (*DHCP*). Although a DHCP packet contains many fields, the basic packet exchange is straightforward. A computer, known as a *host*, broadcasts a DHCP *Discover* message. A DHCP server on the local network replies by sending a DHCP *Offer* message that contains an IP address for the host, a 32-bit subnet mask for the local network, and the address of a default router. The computer replies by sending a *Request* message to the server.

Instead of engaging in a DHCP exchange when the system boots, our code waits until an IP address is needed. An application calls *getlocalip* to obtain the local IP address. If the IP address has been fetched previously, the code merely returns the value. If the host's IP address is unknown, *getlocalip* uses DHCP to obtain the address. The code starts by creating and sending a DHCP *Discover* message. It then uses *udp_recv* to wait for a reply.

File *dhcp.h* defines the structure of a DHCP message. The entire DHCP message will be carried in the payload of a UDP message, which is carried in an IP datagram, which is carried in an Ethernet packet.

```
/* dhcp.h - Definitions related to DHCP */

#define DHCP

#define DHCP_RETRY      5

#define DHCP_PADDING                    0
#define DHCP_SUBNET_MASK                1
#define DHCP_ROUTER                     3
```

†The code can be obtained from the website *xinu.cs.purdue.edu*

```
#define DHCP_DNS_SERVER                 6
#define DHCP_DOMAIN_NAME                15
#define DHCP_VENDER_OPTIONS             43
#define DHCP_REQUESTED_IP               50
#define DHCP_IP_ADDR_LEASE_TIME         51
#define DHCP_OPTION_OVERLOAD            52
#define DHCP_MESSAGE_TYPE               53
#define DHCP_SERVER_ID                  54
#define DHCP_PARAMETER_REQUEST_LIST     55
#define DHCP_MESSAGE                    56
#define DHCP_MAXIMUM_DHCP_MESSAGE_SIZE  57
#define DHCP_RENEWAL_TIME_VALUE         58
#define DHCP_REBINDING_TIME_VALUE       59
#define DHCP_VENDOR_CLASS_ID            60
#define DHCP_CLIENT_ID                  61
#define DHCP_TFTP_SERVER_NAME           66
#define DHCP_BOOTFILE_NAME              67
#define DHCP_CLIENT_SYS_ARCH            93
#define DHCP_CLIENT_NET_ID              94
#define DHCP_CLIENT_MACHINE_ID          97
#define DHCP_MESSAGE_END               255

#pragma pack(2)
struct  dhcpmsg {
        byte    dc_bop;                 /* DHCP bootp op 1=req 2=reply  */
        byte    dc_htype;               /* DHCP hardware type           */
        byte    dc_hlen;                /* DHCP hardware address length */
        byte    dc_hops;                /* DHCP hop count               */
        uint32  dc_xid;                 /* DHCP xid                     */
        uint16  dc_secs;                /* DHCP seconds                 */
        uint16  dc_flags;               /* DHCP flags                   */
        uint32  dc_cip;                 /* DHCP client IP address       */
        uint32  dc_yip;                 /* DHCP your IP address         */
        uint32  dc_sip;                 /* DHCP server IP address       */
        uint32  dc_gip;                 /* DHCP gateway IP address      */
        byte    dc_chaddr[16];          /* DHCP client hardware address */
        union {
                byte    dc_bootp[192];  /* DHCP bootp area (zero)       */
                struct {
                        byte   sname[64];       /* TFTP Server Name     */
                        byte   bootfile[128];   /* TFTP File name       */
                };
        };
        uint32  dc_cookie;              /* DHCP cookie                  */
        byte    dc_opt[1024];           /* DHCP options area (large     */
```

```
                                        /*  enough to hold more than    */
                                        /*  reasonable options          */
};
#pragma pack()
```

If the local IP address has not been initialized, function *getlocalip* creates and sends a DHCP *Discover* message, waits to receive a reply, extracts the IP address, subnet mask, and default router address from the reply and stores them in *Netdata*, and returns the IP address. The code can be found in file *dhcp.c*:

```
/* dhcp.c - getlocalip */

#include <xinu.h>

/*------------------------------------------------------------------------
 * dhcp_get_opt_val  -  Retrieve a pointer to the value for a specified
 *                        DHCP options key
 *------------------------------------------------------------------------
 */
char*   dhcp_get_opt_val(
          const struct dhcpmsg* dmsg,   /* DHCP Message                */
          uint32 dmsg_size,             /* Size of DHCP Message        */
          uint8 option_key              /* Option key to retrieve      */
        )
{
        unsigned char* opt_tmp;
        unsigned char* eom;

        eom = (unsigned char*)dmsg + dmsg_size - 1;
        opt_tmp = (unsigned char*)dmsg->dc_opt;

        while(opt_tmp < eom) {

                /* If the option value matches return the value */

                if((*opt_tmp) == option_key) {

                        /* Offset past the option value and the size    */

                        return (char*)(opt_tmp+2);
                }
                opt_tmp++;      /* Move to length octet */
                opt_tmp += *(uint8*)opt_tmp + 1;
        }
```

```
        /* Option value not found */

        return NULL;
}

/*-----------------------------------------------------------------------
 * dhcp_bld_bootp_msg  -  Set the common fields for all DHCP messages
 *-----------------------------------------------------------------------
 */
void    dhcp_bld_bootp_msg(struct dhcpmsg* dmsg)
{
        uint32  xid;                    /* Xid used for the exchange    */

        memcpy(&xid, NetData.ethucast, 4); /* Use 4 bytes from MAC as   */
                                           /*    unique XID             */
        memset(dmsg, 0x00, sizeof(struct dhcpmsg));

        dmsg->dc_bop = 0x01;            /* Outgoing request             */
        dmsg->dc_htype = 0x01;          /* Hardware type is Ethernet    */
        dmsg->dc_hlen = 0x06;           /* Hardware address length      */
        dmsg->dc_hops = 0x00;           /* Hop count                    */
        dmsg->dc_xid = htonl(xid);      /* Xid (unique ID)              */
        dmsg->dc_secs = 0x0000;         /* Seconds                      */
        dmsg->dc_flags = 0x0000;        /* Flags                        */
        dmsg->dc_cip = 0x00000000;      /* Client IP address            */
        dmsg->dc_yip = 0x00000000;      /* Your IP address              */
        dmsg->dc_sip = 0x00000000;      /* Server IP address            */
        dmsg->dc_gip = 0x00000000;      /* Gateway IP address           */
        memset(&dmsg->dc_chaddr,'\0',16);/* Client hardware address     */
        memcpy(&dmsg->dc_chaddr, NetData.ethucast, ETH_ADDR_LEN);
        memset(&dmsg->dc_bootp,'\0',192);/* Zero the bootp area         */
        dmsg->dc_cookie = htonl(0x63825363); /* Magic cookie for DHCP   */
}

/*-----------------------------------------------------------------------
 * dhcp_bld_disc  -  handcraft a DHCP Discover message in dmsg
 *-----------------------------------------------------------------------
 */
int32   dhcp_bld_disc(struct dhcpmsg* dmsg)
{
        uint32  j = 0;

        dhcp_bld_bootp_msg(dmsg);
```

```
        dmsg->dc_opt[j++] = 0xff & 53;  /* DHCP message type option     */
        dmsg->dc_opt[j++] = 0xff &  1;  /* Option length                */
        dmsg->dc_opt[j++] = 0xff &  1;  /* DHCP Dicover message         */
        dmsg->dc_opt[j++] = 0xff &  0;  /* Options padding              */

        dmsg->dc_opt[j++] = 0xff & 55;  /* DHCP parameter request list  */
        dmsg->dc_opt[j++] = 0xff &  2;  /* Option length                */
        dmsg->dc_opt[j++] = 0xff &  1;  /* Request subnet mask          */
        dmsg->dc_opt[j++] = 0xff &  3;  /* Request default router addr->*/

        return (uint32)((char *)&dmsg->dc_opt[j] - (char *)dmsg + 1);
}

/*-----------------------------------------------------------------------
 * dhcp_bld_req - handcraft a DHCP request message in dmsg
 *-----------------------------------------------------------------------
 */
int32   dhcp_bld_req(
          struct dhcpmsg* dmsg,         /* DHCP message to build        */
          const struct dhcpmsg* dmsg_offer, /* DHCP offer message       */
          uint32 dsmg_offer_size        /* Size of DHCP offer message   */
        )
{
        uint32  j = 0;
        uint32* server_ip;              /* Take the DHCP server IP addr */
                                        /*   from DHCP offer message    */

        dhcp_bld_bootp_msg(dmsg);
        dmsg->dc_sip = dmsg_offer->dc_sip; /* Server IP address         */

        dmsg->dc_opt[j++] = 0xff & 53;  /* DHCP message type option     */
        dmsg->dc_opt[j++] = 0xff &  1;  /* Option length                */
        dmsg->dc_opt[j++] = 0xff &  3;  /* DHCP Request message         */
        dmsg->dc_opt[j++] = 0xff &  0;  /* Options padding              */

        dmsg->dc_opt[j++] = 0xff & 50;  /* Requested IP                 */
        dmsg->dc_opt[j++] = 0xff &  4;  /* Option length                */
        *((uint32*)&dmsg->dc_opt[j]) = dmsg_offer->dc_yip;
        j += 4;

        /* Retrieve the DHCP server IP from the DHCP options */
        server_ip = (uint32*)dhcp_get_opt_val(dmsg_offer,
                                  dsmg_offer_size, DHCP_SERVER_ID);

        if(server_ip == 0) {
```

```
                kprintf("Unable to get server IP add. from DHCP Offer\n");
                return SYSERR;
        }

        dmsg->dc_opt[j++] = 0xff & 54;  /* Server IP                    */
        dmsg->dc_opt[j++] = 0xff &  4;  /* Option length                */
        *((uint32*)&dmsg->dc_opt[j]) = *server_ip;
        j += 4;

        return (uint32)((char *)&dmsg->dc_opt[j] - (char *)dmsg + 1);
}

/*------------------------------------------------------------------------
 * getlocalip - use DHCP to obtain an IP address
 *------------------------------------------------------------------------
 */
uint32  getlocalip(void)
{
        int32   slot;                   /* UDP slot to use              */
        struct  dhcpmsg dmsg_snd;       /* Holds outgoing DHCP messages */
        struct  dhcpmsg dmsg_rvc;       /* Holds incoming DHCP messages */

        int32   i, j;                   /* Retry counters               */
        int32   len;                    /* Length of data sent          */
        int32   inlen;                  /* Length of data received      */
        char    *optptr;                /* Pointer to options area      */
        char    *eop;                   /* Address of end of packet     */
        int32   msgtype;                /* Type of DCHP message         */
        uint32  addrmask;               /* Address mask for network     */
        uint32  routeraddr;             /* Default router address       */
        uint32  tmp;                    /* Used for byte conversion     */
        uint32* tmp_server_ip;          /* Temporary DHCP server pointer*/

        slot = udp_register(0, UDP_DHCP_SPORT, UDP_DHCP_CPORT);
        if (slot == SYSERR) {
                kprintf("getlocalip: cannot register with UDP\n");
                return SYSERR;
        }

        len = dhcp_bld_disc(&dmsg_snd);
        if(len == SYSERR) {
                kprintf("getlocalip: Unable to build DHCP discover\n");
                return SYSERR;
        }
```

```
for (i = 0; i < DHCP_RETRY; i++) {
        udp_sendto(slot, IP_BCAST, UDP_DHCP_SPORT,
                                        (char *)&dmsg_snd, len);

        /* Read 3 incoming DHCP messages and check for an offer */
        /*      or wait for three timeout periods if no message */
        /*      arrives.                                        */

        for (j=0; j<3; j++) {
                inlen = udp_recv(slot, (char *)&dmsg_rvc,
                                        sizeof(struct dhcpmsg),2000);
                if (inlen == TIMEOUT) {
                        continue;
                } else if (inlen == SYSERR) {
                        return SYSERR;
                }
                /* Check that incoming message is a valid       */
                /*   response (ID matches our request)          */

                if (dmsg_rvc.dc_xid != dmsg_snd.dc_xid) {
                        continue;
                }

                eop = (char *)&dmsg_rvc + inlen - 1;
                optptr = (char *)&dmsg_rvc.dc_opt;
                msgtype = addrmask = routeraddr = 0;

                while (optptr < eop) {

                    switch (*optptr) {
                        case 53:        /* Message type */
                                msgtype = 0xff & *(optptr+2);
                        break;

                        case 1:         /* Subnet mask */
                                memcpy((void *)&tmp, optptr+2, 4);
                                addrmask = ntohl(tmp);
                        break;

                        case 3:         /* Router address */
                                memcpy((void *)&tmp, optptr+2, 4);
                                routeraddr = ntohl(tmp);
                                break;
                    }
                    optptr++;   /* Move to length octet */
```

```
    optptr += (0xff & *optptr) + 1;
}

if (msgtype == 0x02) {  /* Offer - send request */
        len = dhcp_bld_req(&dmsg_snd, &dmsg_rvc,
                                        inlen);
        if(len == SYSERR) {
                kprintf("getlocalip: %s\n",
                  "Unable to build DHCP request");
                return SYSERR;
        }
        udp_sendto(slot, IP_BCAST, UDP_DHCP_SPORT,
                        (char *)&dmsg_snd, len);
        continue;

} else if (dmsg_rvc.dc_opt[2] != 0x05) {
        /* If not an ack skip it */
        continue;
}
if (addrmask != 0) {
        NetData.ipmask = addrmask;
}
if (routeraddr != 0) {
        NetData.iprouter = routeraddr;
}
NetData.ipucast = ntohl(dmsg_rvc.dc_yip);
NetData.ipprefix = NetData.ipucast &
                                NetData.ipmask;
NetData.ipbcast = NetData.ipprefix |
                                ~NetData.ipmask;
NetData.ipvalid = TRUE;
udp_release(slot);

/* Retrieve the boot server IP */
if(dot2ip((char*)dmsg_rvc.sname,
                    &NetData.bootserver) != OK) {

  /* Could not retrieve the boot server from    */
  /*  the  BOOTP fields, so use the DHCP server */
  /*  address                                   */
  tmp_server_ip = (uint32*)dhcp_get_opt_val(
                &dmsg_rvc, len, DHCP_SERVER_ID);
  if(tmp_server_ip == 0) {
    kprintf("Cannot retrieve boot server addr\n");
        return (uint32)SYSERR;
```

```
                        }
                    NetData.bootserver = ntohl(*tmp_server_ip);
                    }
                    memcpy(NetData.bootfile, dmsg_rvc.bootfile,
                                        sizeof(dmsg_rvc.bootfile));
                    return NetData.ipucast;
                }
        }

        kprintf("DHCP failed to get response\n");
        udp_release(slot);
        return (uint32)SYSERR;
}
```

A DHCP server responds to the initial *Discover* message by sending the requested information, and the client responds with a *Request*. When it receives a reply, *getlocalip* examines the options area of the message. DHCP is unusual because the options carry important information. In particular, the type of the DHCP message is stored in the options area as well as information that a computer system uses to initialize network parameters. Three options are crucial to our implementation: option 53 defines the *type* of a DHCP message, option 1 specifies a subnet mask used on the local network, and option 3 specifies the address of a default router. If the options are present, *getlocalip* can extract the needed information from the reply, store the information for successive calls, and return the IP address to the caller.

Further details of DHCP are beyond the scope of this text. However, it is sufficient to understand that the DHCP code uses the UDP interface exactly the same way that other applications use it. That is, before communication can begin, *getlocalip* must call *udp_register* to register the port that DHCP will use. Once the port has been registered, *getlocalip* can create a DHCP *Discover* message and call *udp_sendto* to broadcast the message. The DHCP *Discover* message causes a DHCP server to respond, and the code can use *udp_recv* to obtain the response, which contains the computer's IP address.

17.14 Perspective

The implementation of Internet protocols described in this chapter is indeed minimalistic. Many details have been omitted, and the code takes many shortcuts. For example, the structures used to define message formats combine multiple layers of the protocol stack and assume that the underlying network is always an Ethernet. More important, the code presented here omits TCP, the major transport protocol used in the Internet. The code also omits IP version 6, the new version of the Internet Protocol that is much more complex than version 4. Unlike IPv4, for example, IPv6 uses variable-size headers, which makes it impossible to use a struct to specify IPv6 datagrams. There-

fore, you should not view the code as typical of a protocol implementation, nor should you assume that the same structure will suffice for a more extensive protocol stack.

Despite its limitations, the code does illustrate the importance of timed operations. In particular, the availability of a timed receive function simplifies the overall structure of the code and makes the operation much easier to understand. If the system did not provide a timed receive, multiple processes would be needed — one process would implement a timer function and another would handle responses.

17.15 Summary

Even small embedded systems use Internet protocols to communicate. As a consequence, most operating systems include software known as a protocol stack.

The chapter examines a minimal protocol stack that supports limited versions of IP, UDP, ICMP, ARP, and DHCP running over an Ethernet. The above protocols are closely interrelated. Both ICMP and UDP messages travel in an IP datagram; a DHCP message travels in a UDP packet.

To accommodate asynchronous packet arrivals, our protocol implementation uses a network input process, *netin*. The *netin* process repeatedly reads an Ethernet packet, validates header fields, and uses header information to determine how to process the packet. When an ARP packet arrives, *netin* calls *arp_in* to handle the packet, when a UDP packet arrives, *netin* calls *udp_in*, and when an ICMP packet arrives, *netin* calls *icmp_in*. For all other packets, *netin* ignores the packet. When receiving packets, our implementation allows a process to specify the maximum time to wait for a packet to arrive. The timeout mechanism can be used to implement retransmission: if a response does not arrive before the timeout occurs, a process can retransmit the request.

EXERCISES

17.1 Can you rewrite the code to eliminate the need for a separate IP output process? Hint: keep a queue of outgoing IP packets with each ARP table entry and arrange for the packets to be sent once an ARP reply arrives.

17.2 Assume you have access to a pair of Xinu systems that connect to the Internet. Use the UDP functions to write two programs that communicate by sending UDP messages to each other.

17.3 To ensure that ARP information does not become stale, the ARP protocol requires an entry to be removed from the cache after a fixed time (even if the entry has been used). Rewrite the code to remove entries that are more than 5 minutes old without using an extra process.

17.4 Redesign *udp_recv* to use timer processes instead of *recvtime*. How many processes are needed? Explain.

17.5 Instead of structuring network code around a *netin* process and an *ipout* process, some operating systems use *software interrupts*. Redesign the networking code in this chapter to use software interrupts.

17.6 If a computer connects to two separate networks (e.g., a Wi-Fi network as well as an Ethernet), what major change is required in the structure of the network code? Hint: how many processes are needed?

17.7 Xinu uses a device paradigm for abstract devices as well as hardware devices. Rewrite the UDP code to use a device paradigm in which a process calls *open* on a UDP master device to specify protocol port and IP address information, and receives the descriptor of a pseudo-device that can be used for communication.

17.8 Can the device paradigm described in the previous exercise handle all of ICMP? Does the answer change if the question is restricted to ICMP echo (i.e., *ping*)? Explain.

17.9 Wireless routers used in residences and dorm rooms pass Internet packets between an Ethernet connection and a Wi-Fi connection. Can you use the code in this chapter to build a wireless router? Why or why not?

Chapter Contents

18.1 Introduction, 417
18.2 The Disk Abstraction, 417
18.3 Operations A Disk Driver Supports, 418
18.4 Block Transfer And High–level I/O Functions, 418
18.5 A Remote Disk Paradigm, 419
18.6 The Important Concept Of Caching, 420
18.7 Semantics Of Disk Operations, 421
18.8 Definition Of Driver Data Structures, 421
18.9 Driver Initialization (rdsinit), 427
18.10 The Upper–half Open Function (rdsopen), 430
18.11 The Remote Communication Function (rdscomm), 432
18.12 The Upper–half Write Function (rdswrite), 435
18.13 The Upper–half Read Function (rdsread), 438
18.14 Flushing Pending Requests, 442
18.15 The Upper–half Control Function (rdscontrol), 442
18.16 Allocating A Disk Buffer (rdsbufalloc), 445
18.17 The Upper–half Close Function (rdsclose), 447
18.18 The Lower–half Communication Process (rdsprocess), 448
18.19 Perspective, 453
18.20 Summary, 454

18

A Remote Disk Driver

For my purpose holds ... To strive, to seek, to find, and not to yield.

— Alfred, Lord Tennyson

18.1 Introduction

Earlier chapters explain I/O devices and the structure of a device driver. Chapter 16 describes how block-oriented devices use DMA, and shows an example Ethernet driver.

This chapter considers the design of a device driver for secondary storage devices known as *disks* or *hard drives*. The chapter focuses on basic data transfer operations. The next chapter describes how higher levels of the system use disk hardware to provide *files* and *directories*.

18.2 The Disk Abstraction

Disk hardware provides a basic abstraction in which a disk is a storage mechanism that has the following properties:

- *Nonvolatile*: stored data persists even if power is removed.
- *Block-oriented*: the hardware can only fetch or store fixed-size blocks.
- *Multi-use*: a given block can be read and written many times.
- *Random-access*: blocks can be read or written in any order.

Like the Ethernet hardware described in Chapter 16, disk hardware typically uses *Direct-Memory-Access* (*DMA*) to allow the disk to transfer an entire block before interrupting the processor. Also like the Ethernet driver, a disk driver does not understand or examine the contents of data blocks. Instead, the driver merely treats the entire storage device as an array of data blocks.

18.3 Operations A Disk Driver Supports

At the device driver level, a disk consists of fixed-size data blocks that can be accessed randomly using three basic operations:

- *Fetch*: Copy the contents of a specified block from the disk to a buffer in memory.
- *Store*: Copy the contents of a memory buffer to a specified block on the disk.
- *Seek*: Move to a specified block on the disk. The *seek* option is only used on electro-mechanical devices (i.e., a magnetic disk) because it provides an optimization that positions the disk head where it will be needed in the future. Thus, when solid state disk technology is used, *seek* is unimportant.

The block size of a disk is derived from the size of a *sector* on magnetic disks. The industry has settled on a de facto standard block size of 512 bytes; throughout the chapter, we will assume 512-byte blocks.†

18.4 Block Transfer And High–level I/O Functions

Because the hardware only provides block transfer, it makes sense to design an interface in which *read* and *write* operations transfer an entire block. The question becomes how to include a block specification in the existing high-level I/O operations. We might require a programmer to call *seek* to move to a specific block before calling *read* or *write* to access data in the block. Unfortunately, requiring a user to call *seek* before each data transfer is clumsy and error prone. Therefore, to keep the interface simple, we will stretch the usual meaning of arguments to *read* and *write*: instead of interpreting the third argument as a buffer size, we will assume the buffer is large enough to hold a disk block, and use the third argument to specify a block number. For example, the call:

```
read(DISK0, buff, 5)
```

requests the driver to read block five from the disk into memory starting at location *buff*.

†Although modern disks often use a larger underlying block size (e.g., 4K bytes), the hardware presents an interface that uses 512-byte blocks.

Our driver will supply two basic operations: *read*, which copies a single block from the disk into memory, and *write*, which copies data from memory onto a specified disk block. In addition, our driver will supply *control* functions that can be used to erase a disk (i.e., destroy all saved data) and to synchronize *write* requests (i.e., ensure that all cached data is written to the disk). The concept of caching and its importance for efficient disk access is discussed later in the chapter.

18.5 A Remote Disk Paradigm

Unlike a traditional computer that has a *local disk*, cloud computing follows a broader notion in which disk storage is separated from processors. Separation allows a physical disk to be accessed from multiple processors, and supports virtual machine migration. Many modern embedded systems follow the cloud approach. To see how disk storage can be separated from processing, we will examine a *remote disk* paradigm. Our remote disk system will provide the same abstraction as a local disk by allowing processes to read and write disk blocks. Instead of using disk hardware, however, our system will send requests over a network to a remote disk server that runs on another computer.

The driver software for the remote disk is surprisingly similar to driver software for a local disk. Overall, both local and remote disk drivers follow the same general organization. That is, the driver functions are partitioned into an upper half and a lower half that communicate through a shared data structure. The main difference arises in the way the lower half operates. Instead of using DMA hardware and interrupts in the lower half, our remote disk driver uses a high-priority communication process. The communication process sends requests over a network to a remote server, and receives responses from the server. Figure 18.1 illustrates the organization.

Figure 18.1 Organization of a remote disk driver.

18.6 The Important Concept Of Caching

Three key facts help operating system designers optimize disk drivers. First, the underlying disk hardware can only transfer complete blocks. Second, as we will see in the next chapter, a file system allows an application to read and write arbitrary amounts of data. Third, disk hardware is much slower than processor hardware. The consequence of the above three facts should be clear: if a driver attempts to transfer data each time an application reads or writes a few bytes, disk access will become a bottleneck in the system.

To make disk accesses efficient, a driver must use *caching*. That is, when a block is accessed, the driver places a copy of the block in a cache in memory. Subsequent *read* and *write* operations manipulate the cached copy without waiting for I/O operations to access the underlying disk. Eventually, of course, changes must be written back to the disk. Without caching, disk access is intolerably slow, but caching can make the effective performance of a disk system fast enough to accommodate most needs. We can summarize:

> *Because disk access is slow and file system functions often read or write partial blocks, a disk driver must cache copies of blocks to achieve high performance.*

Our example driver illustrates the idea. The shared data structure contains two key items.

- A cache of recently accessed blocks
- A list of pending requests

A cache of recently accessed blocks. Our cache stores recently accessed disk blocks. A block is cached whenever a *read* or *write* operation occurs. Furthermore, when it receives a subsequent operation for a block, the driver always searches the cache before requesting a transfer; the driver does not distinguish between *read* and *write* requests. Thus, if a file system first writes data to a given block and then reads data from the same block, the *read* operation will use the cached copy.

A list of pending requests. Like a traditional driver, our remote disk system allows multiple processes to access a disk, and implements synchronous *read* operations and asynchronous *write* operations. That is, when it reads a block, a process must wait until the data has been fetched. When it writes a block, however, a process does not block — the driver places a copy of the outgoing disk block on a request list, and allows the process to continue execution. The lower-half process, which handles communication with the remote server, continually takes the next item off the request queue and performs the specified operation. Thus, at some time in the future, data will be written to the disk.

18.7 Semantics Of Disk Operations

Although it can employ optimizations such as caching and delayed *write* operations, a disk driver must always preserve the appearance of a synchronous interface. That is, the driver must always return the data that was written most recently.

If we envision the *read* and *write* operations on a given block, they form a sequence:

$$op_1,\ op_2,\ op_3, \ldots\ op_n$$

If op_t is a *read* operation for block *i*, the driver must deliver the data that was written to the block in op_k, where *k* is the highest index of a *write* operation less than *t* (i.e., all the operations between op_k and op_t are *read* operations). To complete the definition, we assume an implicit *write* occurs at time zero before the system starts. Thus, if a system attempts to *read* a block before any calls to *write* the block, the driver returns whatever data was on the disk when the system booted.

We use the term *last-write semantics* to capture the concept, and insist that:

> *A disk driver can use techniques such as caching to optimize performance provided the driver guarantees last-write semantics.*

The example driver uses a FIFO queue to enforce last-write semantics:

> *Items are inserted at the tail of the request queue; the lower-half process continually selects and performs the item at the head of the queue.*

Because items are always inserted at the tail, the driver handles requests in the same order they were made. Thus, if process A *reads* block five and process B *writes* block five later, the two requests will appear in the correct order in the queue. The *read* request will be serviced first, followed by the *write* request. We will see that the queue discipline is extended to the cache: driver functions always start at the head when searching the cache. Our code relies on the queue discipline to ensure that a process receives data according to last-write semantics.

18.8 Definition Of Driver Data Structures

File *rdisksys.h* defines the constants and data structures used by the remote disk system. The file defines the format of a disk buffer. Each buffer includes a header that specifies the number of the disk block stored in the buffer and fields that are used to link the buffer onto the request list, cache, or free list. In addition, the file defines the contents of the device control block and the format of messages sent to the remote server.

```
/* rdisksys.h - definitions for remote disk system pseudo-devices */

#ifndef Nrds
#define Nrds            1
#endif

/* Remote disk block size */

#define RD_BLKSIZ       512

/* Global data for the remote disk server */

#ifndef RD_SERVER_IP
#define RD_SERVER_IP    "255.255.255.255"
#endif

#ifndef RD_SERVER_PORT
#define RD_SERVER_PORT  33124
#endif

#ifndef RD_LOC_PORT
#define RD_LOC_PORT     33124           /* Base port number - minor dev */
                                        /*   number is added to insure  */
                                        /*   that each device is unique */
#endif

/* Control block for remote disk device */

#define RD_IDLEN        64              /* Size of a remote disk ID     */
#define RD_BUFFS        64              /* Number of disk buffers       */
#define RD_STACK        16384           /* Stack size for comm. process */
#define RD_PRIO         200             /* Priorty of comm. process     */

/* Constants for state of the device */

#define RD_FREE          0              /* Device is available          */
#define RD_OPEN          1              /* Device is open (in use)      */
#define RD_PEND          2              /* Open is pending              */

/* Operations for request queue */

#define RD_OP_READ      1               /* Read operation on req. list  */
#define RD_OP_WRITE     2               /* Write operation on req. list */
#define RD_OP_SYNC      3               /* Sync operation on req. list  */

/* Status values for a buffer */
```

```
#define RD_VALID        0               /* Buffer contains valid data   */
#define RD_INVALID      1               /* Buffer does not contain data */

/* Definition of a buffer with a header that allows the same node to be */
/*  used as a request on the request queue, an item in the cache, or a  */
/*  node on the free list of buffers                                    */

struct  rdbuff  {                       /* Request list node            */
        struct  rdbuff  *rd_next;       /* Ptr to next node on a list   */
        struct  rdbuff  *rd_prev;       /* Ptr to prev node on a list   */
        int32   rd_op;                  /* Operation - read/write/sync  */
        int32   rd_refcnt;              /* Reference count of processes */
                                        /*   reading the block          */
        uint32  rd_blknum;              /* Block number of this block   */
        int32   rd_status;              /* Is buffer currently valid?   */
        pid32   rd_pid;                 /* Process that initiated a     */
                                        /*   read request for the block */
        char    rd_block[RD_BLKSIZ];    /* Space to hold one disk block */
};

struct  rdscblk {
        int32   rd_state;               /* State of device              */
        char    rd_id[RD_IDLEN];        /* Disk ID currently being used */
        int32   rd_seq;                 /* Next sequence number to use  */
        /* Request queue head and tail */
        struct  rdbuff  *rd_rhnext;     /* Head of request queue: next  */
        struct  rdbuff  *rd_rhprev;     /*   and previous               */
        struct  rdbuff  *rd_rtnext;     /* Tail of request queue: next  */
        struct  rdbuff  *rd_rtprev;     /*   (null) and previous        */

        /* Cache head and tail */

        struct  rdbuff  *rd_chnext;     /* Head of cache: next and      */
        struct  rdbuff  *rd_chprev;     /*   previous                   */
        struct  rdbuff  *rd_ctnext;     /* Tail of cache: next (null)   */
        struct  rdbuff  *rd_ctprev;     /*   and previous               */

        /* Free list head (singly-linked) */

        struct  rdbuff  *rd_free;       /* Pointer to free list         */

        pid32   rd_comproc;             /* Process ID of comm. process  */
        bool8   rd_comruns;             /* Has comm. process started?   */
        sid32   rd_availsem;            /* Semaphore ID for avail buffs */
        sid32   rd_reqsem;              /* Semaphore ID for requests    */
```

```
        uint32  rd_ser_ip;              /* Server IP address            */
        uint16  rd_ser_port;            /* Server UDP port              */
        uint16  rd_loc_port;            /* Local (client) UPD port      */
        bool8   rd_registered;          /* Has UDP port been registered?*/
        int32   rd_udpslot;             /* Registered UDP slot          */
};


extern  struct  rdscblk rdstab[];       /* Remote disk control block    */

/* Definitions of parameters used during server access */

#define RD_RETRIES      3               /* Times to retry sending a msg */
#define RD_TIMEOUT      2000            /* Timeout for reply (2 seconds)*/

/* Control functions for a remote file pseudo device */

#define RDS_CTL_DEL     1               /* Delete (erase) an entire disk*/
#define RDS_CTL_SYNC    2               /* Write all pending blocks     */

/************************************************************************/
/*      Definition of messages exchanged with the remote disk server    */
/************************************************************************/
/* Values for the type field in messages */

#define RD_MSG_RESPONSE 0x0100          /* Bit that indicates response  */

#define RD_MSG_RREQ     0x0010          /* Read request and response    */
#define RD_MSG_RRES     (RD_MSG_RREQ | RD_MSG_RESPONSE)

#define RD_MSG_WREQ     0x0020          /* Write request and response   */
#define RD_MSG_WRES     (RD_MSG_WREQ | RD_MSG_RESPONSE)

#define RD_MSG_OREQ     0x0030          /* Open request and response    */
#define RD_MSG_ORES     (RD_MSG_OREQ | RD_MSG_RESPONSE)

#define RD_MSG_CREQ     0x0040          /* Close request and response   */
#define RD_MSG_CRES     (RD_MSG_CREQ | RD_MSG_RESPONSE)

#define RD_MSG_DREQ     0x0050          /* Delete request and response  */
#define RD_MSG_DRES     (RD_MSG_DREQ | RD_MSG_RESPONSE)

#define RD_MIN_REQ      RD_MSG_RREQ     /* Minimum request type         */
#define RD_MAX_REQ      RD_MSG_DREQ     /* Maximum request type         */
```

```
/* Message header fields present in each message */

#define RD_MSG_HDR                      /* Common message fields        */\
        uint16  rd_type;                /* Message type                 */\
        uint16  rd_status;              /* 0 in req, status in response */\
        uint32  rd_seq;                 /* Message sequence number      */\
        char    rd_id[RD_IDLEN];        /* Null-terminated disk ID      */

/************************************************************************/
/*                              Header                                  */
/************************************************************************/
/* The standard header present in all messages with no extra fields */
#pragma pack(2)
struct  rd_msg_hdr {                    /* Header fields present in each*/
        RD_MSG_HDR                      /*   remote file system message */
};
#pragma pack()

/************************************************************************/
/*                              Read                                    */
/************************************************************************/
#pragma pack(2)
struct  rd_msg_rreq     {               /* Remote file read request     */
        RD_MSG_HDR                      /* Header fields                */
        uint32  rd_blk;                 /* Block number to read         */
};
#pragma pack()

#pragma pack(2)
struct  rd_msg_rres     {               /* Remote file read reply       */
        RD_MSG_HDR                      /* Header fields                */
        uint32  rd_blk;                 /* Block number that was read   */
        char    rd_data[RD_BLKSIZ];     /* Array containing one block   */
};
#pragma pack()

/************************************************************************/
/*                              Write                                   */
/************************************************************************/
#pragma pack(2)
struct  rd_msg_wreq     {               /* Remote file write request    */
        RD_MSG_HDR                      /* Header fields                */
        uint32  rd_blk;                 /* Block number to write        */
        char    rd_data[RD_BLKSIZ];     /* Array containing one block   */
};
#pragma pack()
```

```
#pragma pack(2)
struct  rd_msg_wres     {               /* Remote file write response   */
        RD_MSG_HDR                      /* Header fields                */
        uint32  rd_blk;                 /* Block number that was written*/
};
#pragma pack()

/***********************************************************************/
/*                              Open                                   */
/***********************************************************************/
#pragma pack(2)
struct  rd_msg_oreq     {               /* Remote file open request     */
        RD_MSG_HDR                      /* Header fields                */
};
#pragma pack()

#pragma pack(2)
struct  rd_msg_ores     {               /* Remote file open response    */
        RD_MSG_HDR                      /* Header fields                */
};
#pragma pack()

/***********************************************************************/
/*                              Close                                  */
/***********************************************************************/
#pragma pack(2)
struct  rd_msg_creq     {               /* Remote file close request    */
        RD_MSG_HDR                      /* Header fields                */
};
#pragma pack()

#pragma pack(2)
struct  rd_msg_cres     {               /* Remote file close response   */
        RD_MSG_HDR                      /* Header fields                */
};
#pragma pack()

/***********************************************************************/
/*                              Delete                                 */
/***********************************************************************/
#pragma pack(2)
struct  rd_msg_dreq     {               /* Remote file delete request   */
        RD_MSG_HDR                      /* Header fields                */
};
#pragma pack()
```

```
#pragma pack(2)
struct  rd_msg_dres     {               /* Remote file delete response  */
        RD_MSG_HDR                      /* Header fields                */
};
#pragma pack()
```

18.9 Driver Initialization (rdsinit)

Although initialization is completed after the other pieces of the driver have been designed, we have chosen to examine the initialization function now because it will help us understand the shared data structures. File *rdsinit.c* contains the driver initialization code:

```
/* rdsinit.c - rdsinit */

#include <xinu.h>

struct  rdscblk rdstab[Nrds];

/*------------------------------------------------------------------------
 *  rdsinit  -  Initialize the remote disk system device
 *------------------------------------------------------------------------
 */
devcall rdsinit (
          struct dentry *devptr         /* Entry in device switch table */
        )
{
        struct  rdscblk *rdptr;         /* Ptr to device contol block   */
        struct  rdbuff  *bptr;          /* Ptr to buffer in memory      */
                                        /*   used to form linked list   */
        struct  rdbuff  *pptr;          /* Ptr to previous buff on list */
        struct  rdbuff  *buffend;       /* Last address in buffer memory*/
        uint32  size;                   /* Total size of memory needed  */
                                        /*   buffers                    */

        /* Obtain address of control block */

        rdptr = &rdstab[devptr->dvminor];

        /* Set control block to unused */

        rdptr->rd_state = RD_FREE;
        rdptr->rd_id[0] = NULLCH;
```

```
/* Set initial message sequence number */

rdptr->rd_seq = 1;

/* Initialize request queue and cache to empty */

rdptr->rd_rhnext = (struct rdbuff *) &rdptr->rd_rtnext;
rdptr->rd_rhprev = (struct rdbuff *)NULL;

rdptr->rd_rtnext = (struct rdbuff *)NULL;
rdptr->rd_rtprev = (struct rdbuff *) &rdptr->rd_rhnext;


rdptr->rd_chnext = (struct rdbuff *) &rdptr->rd_ctnext;
rdptr->rd_chprev = (struct rdbuff *)NULL;

rdptr->rd_ctnext = (struct rdbuff *)NULL;
rdptr->rd_ctprev = (struct rdbuff *) &rdptr->rd_chnext;

/* Allocate memory for a set of buffers (actually request       */
/*    blocks and link them to form the initial free list        */

size = sizeof(struct rdbuff) * RD_BUFFS;

bptr = (struct rdbuff *)getmem(size);
rdptr->rd_free = bptr;

if ((int32)bptr == SYSERR) {
        panic("Cannot allocate memory for remote disk buffers");
}

buffend = (struct rdbuff *) ((char *)bptr + size);
while (bptr < buffend) {        /* walk through memory */
        pptr = bptr;
        bptr = (struct rdbuff *)
                        (sizeof(struct rdbuff)+ (char *)bptr);
        pptr->rd_status = RD_INVALID;   /* Buffer is empty      */
        pptr->rd_next = bptr;           /* Point to next buffer */
}
pptr->rd_next = (struct rdbuff *) NULL; /* Last buffer on list  */

/* Create the request list and available buffer semaphores */

rdptr->rd_availsem = semcreate(RD_BUFFS);
rdptr->rd_reqsem   = semcreate(0);
```

```
        /* Set the server IP address, server port, and local port */

        if ( dot2ip(RD_SERVER_IP, &rdptr->rd_ser_ip) == SYSERR ) {
                panic("invalid IP address for remote disk server");
        }

        /* Set the port numbers */

        rdptr->rd_ser_port = RD_SERVER_PORT;
        rdptr->rd_loc_port = RD_LOC_PORT + devptr->dvminor;

        /* Specify that the server port is not yet registered */

        rdptr->rd_registered = FALSE;

        /* Create a communication process */

        rdptr->rd_comproc = create(rdsprocess, RD_STACK, RD_PRIO,
                                            "rdsproc", 1, rdptr);

        if (rdptr->rd_comproc == SYSERR) {
                panic("Cannot create remote disk process");
        }
        resume(rdptr->rd_comproc);

        return OK;
}
```

In addition to initializing data structures, *rdsinit* performs three important tasks. It allocates a set of disk buffers and links them onto the free list, it creates two semaphores that control processing, and it creates the high-priority process that communicates with the server.

One of the semaphores, *rd_reqsem*, guards the request list. The semaphore starts with count zero, and is signaled each time a new request is added to the request queue. The communication process waits on *rd_reqsem* before extracting an item from the list, which means the process will block if the list is empty. Because the semaphore starts with an initial count of zero, the communication process will block on the semaphore until a request has been deposited in the queue.

The other semaphore, *rd_availsem*, counts the number of buffers that are available for use (i.e., free or in the cache). Initially, *RD_BUFFS* buffers are on the free list and *rd_availsem* has a count equal to *RD_BUFFS*. When a buffer is needed, a caller waits on the semaphore. The cache poses a special case because not all buffers in the cache are available. Buffers for which operations have been completed can be taken from the cache at any time. However, buffers that correspond to pending operations (e.g., a

process has requested a *read* operation but has not yet extracted the data) must remain in the cache. We will see how the cache and the semaphores are used later.

18.10 The Upper–half Open Function (rdsopen)

The remote disk server allows multiple clients to access the server simultaneously. Each client supplies a unique identification string which allows the server to distinguish among clients. Instead of using a hardware value (e.g., an Ethernet address) as the unique string, the example code allows a user to specify the ID string by calling *open* on the disk device. The chief advantage of separating the ID from the hardware is portability — the remote disk ID can be bound to the operating system image, which means that moving the image from one physical computer to another does not change the disk that the system is using.

When a process calls *open* for a remote disk device, the second argument is interpreted as an ID string. The string is copied into the device control block, and the same ID is used as long as the device remains open. It is possible to close the remote disk device and reopen it with a new ID (i.e., connect to a different remote disk). However, most systems are expected to open a remote disk device once and never close it. File *rdsopen.c* contains the code:

```
/* rdsopen.c - rdsopen */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rdsopen  -  Open a remote disk device and specify an ID to use
 *------------------------------------------------------------------------
 */

devcall rdsopen (
         struct dentry  *devptr,        /* Entry in device switch table */
         char   *diskid,                /* Disk ID to use               */
         char   *mode                   /* Unused for a remote disk     */
        )
{
        struct  rdscblk *rdptr;         /* Ptr to control block entry   */
        struct  rd_msg_oreq msg;        /* Message to be sent           */
        struct  rd_msg_ores resp;       /* Buffer to hold response      */
        int32   retval;                 /* Return value from rdscomm    */
        int32   len;                    /* Counts chars in diskid       */
        char    *idto;                  /* Ptr to ID string copy        */
        char    *idfrom;                /* Pointer into ID string       */
```

```
rdptr = &rdstab[devptr->dvminor];

/* Reject if device is already open */

if (rdptr->rd_state != RD_FREE) {
        return SYSERR;
}
rdptr->rd_state = RD_PEND;

/* Copy disk ID into free table slot */

idto = rdptr->rd_id;
idfrom = diskid;
len = 0;
while ( (*idto++ = *idfrom++) != NULLCH) {
        len++;
        if (len >= RD_IDLEN) {  /* ID string is too long */
                return SYSERR;
        }
}

/* Verify that name is non-null */

if (len == 0) {
        return SYSERR;
}

/* Hand-craft an open request message to be sent to the server */

msg.rd_type = htons(RD_MSG_OREQ);/* Request an open             */
msg.rd_status = htons(0);
msg.rd_seq = 0;                  /* Rdscomm fills in an entry    */
idto = msg.rd_id;
memset(idto, NULLCH, RD_IDLEN);/* initialize ID to zero bytes   */

idfrom = diskid;
while ( (*idto++ = *idfrom++) != NULLCH ) { /* Copy ID to req.  */
        ;
}

/* Send message and receive response */

retval = rdscomm((struct rd_msg_hdr *)&msg,
                                 sizeof(struct rd_msg_oreq),
                 (struct rd_msg_hdr *)&resp,
```

```
                                        sizeof(struct rd_msg_ores),
                                rdptr );

        /* Check response */

        if (retval == SYSERR) {
                rdptr->rd_state = RD_FREE;
                return SYSERR;
        } else if (retval == TIMEOUT) {
                kprintf("Timeout during remote file open\n\r");
                rdptr->rd_state = RD_FREE;
                return SYSERR;
        } else if (ntohs(resp.rd_status) != 0) {
                rdptr->rd_state = RD_FREE;
                return SYSERR;
        }

        /* Change state of device to indicate currently open */

        rdptr->rd_state = RD_OPEN;

        /* Return device descriptor */

        return devptr->dvnum;
}
```

18.11 The Remote Communication Function (rdscomm)

As one of the steps in opening the local remote disk device, *rdsopen* exchanges a message with the remote server. It places an *open request* message in local variable *msg*, and calls *rdscomm* to forward the message to the server. *Rdscomm* takes arguments that specify an outgoing message, a buffer for a reply, and the length of each. It sends the outgoing message to the server, and awaits a reply. If the reply is valid, *rdscomm* returns the length of the reply to the caller; otherwise, it returns *SYSERR* to indicate that an error occurred or *TIMEOUT* to indicate that no response was received. File *rdscomm.c* contains the code:

```
/* rdscomm.c - rdscomm */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rdscomm  -  handle communication with a remote disk server (send a
 *               request and receive a reply, including sequencing and
 *               retries)
```

```
 *------------------------------------------------------------------------
 */
status  rdscomm (
          struct rd_msg_hdr *msg,       /* Message to send              */
          int32             mlen,       /* Message length               */
          struct rd_msg_hdr *reply,     /* Buffer for reply             */
          int32             rlen,       /* Size of reply buffer         */
          struct rdscblk    *rdptr      /* Ptr to device control block  */
        )
{
        int32   i;                      /* Counts retries               */
        int32   retval;                 /* Return value                 */
        int32   seq;                    /* Sequence for this exchange   */
        uint32  localip;                /* Local IP address             */
        int16   rtype;                  /* Reply type in host byte order*/
        bool8   xmit;                   /* Should we transmit again?    */
        int32   slot;                   /* UDP slot                     */

        /* For the first time after reboot, register the server port */

        if ( ! rdptr->rd_registered ) {
                slot = udp_register(0, rdptr->rd_ser_port,
                                rdptr->rd_loc_port);
                if(slot == SYSERR) {
                        return SYSERR;
                }
                rdptr->rd_udpslot = slot;
                rdptr->rd_registered = TRUE;
        }

        if ( NetData.ipvalid == FALSE ) {
                localip = getlocalip();
                if((int32)localip == SYSERR) {
                        return SYSERR;
                }
        }

        /* Retrieve the saved UDP slot number  */

        slot = rdptr->rd_udpslot;

        /* Assign message next sequence number */

        seq = rdptr->rd_seq++;
        msg->rd_seq = htonl(seq);
```

```
/* Repeat RD_RETRIES times: send message and receive reply */

xmit = TRUE;
for (i=0; i<RD_RETRIES; i++) {
    if (xmit) {

        /* Send a copy of the message */

        retval = udp_sendto(slot, rdptr->rd_ser_ip, rdptr->rd_ser_port,
                                (char *)msg, mlen);
        if (retval == SYSERR) {
                kprintf("Cannot send to remote disk server\n\r");
                return SYSERR;
        }
    } else {
        xmit = TRUE;
    }

    /* Receive a reply */

    retval = udp_recv(slot, (char *)reply, rlen,
                                        RD_TIMEOUT);

    if (retval == TIMEOUT) {
        continue;
    } else if (retval == SYSERR) {
        kprintf("Error reading remote disk reply\n\r");
        return SYSERR;
    }

    /* Verify that sequence in reply matches request */


    if (ntohl(reply->rd_seq) < seq) {
        xmit = FALSE;
    } else if (ntohl(reply->rd_seq) != seq) {
                continue;
    }

    /* Verify the type in the reply matches the request */

    rtype = ntohs(reply->rd_type);
    if (rtype != ( ntohs(msg->rd_type) | RD_MSG_RESPONSE) ) {
        continue;
    }
```

```
            /* Check the status */

            if (ntohs(reply->rd_status) != 0) {
                return SYSERR;
            }

            return OK;
        }

        /* Retries exhausted without success */

        kprintf("Timeout on exchange with remote disk server\n\r");
        return TIMEOUT;
}
```

Rdscomm uses UDP for communication with the remote server.† The code takes two steps related to the use of UDP. First, *rdscomm* checks to see whether the UDP port has already been registered, and calls *udp_register* if it has not. It may seem that the check is unnecessary because the port will not be registered until *rdscomm* runs. However, checking at runtime allows the remote disk system to be restarted. Second, *rdscomm* checks to see whether the computer has already obtained an IP address (which is required for Internet communication). If an address has not been assigned, *rdscomm* calls *getlocalip* to obtain an address. Once the two steps are complete, *rdscomm* is ready to communicate with the remote disk server.

Rdscomm assigns the next sequence number to the message, and enters a loop that iterates *RD_RETRIES* times. On each iteration, *rdscomm* calls *udp_sendto* to transmit a copy of the message to the server, and calls *udp_recv* to receive a reply. If a reply arrives, *rdscomm* verifies that the sequence number of the reply matches the sequence number of the request that was sent, that the *type* of the reply matches the type of the request, and that the status value indicates success (i.e., is zero). If the reply is valid, *rdscomm* returns *OK* to the caller; otherwise, it returns an error indication.

18.12 The Upper–half Write Function (rdswrite)

Because the remote disk system provides asynchronous *write* operations, the upper-half *write* function is easiest to understand. The general idea is that a *write* request must be created, the data to be written must be copied into the request, and the request must be enqueued on the request queue. However, because our driver has both a queue of impending requests and a cache of recently-accessed blocks, the driver must handle the case where a request refers to a block that is already in memory on the request queue or in the cache. File *rdswrite.c* contains the code:

†A description of UDP is given in Chapter 17, starting on page 358.

```
/* rdswrite.c - rdswrite */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rdswrite  -  Write a block to a remote disk
 *------------------------------------------------------------------------
 */
devcall rdswrite (
          struct dentry *devptr,        /* Entry in device switch table */
          char  *buff,                  /* Buffer that holds a disk blk */
          int32 blk                     /* Block number to write        */
        )
{
        struct  rdscblk *rdptr;         /* Pointer to control block     */
        struct  rdbuff  *bptr;          /* Pointer to buffer on a list  */
        struct  rdbuff  *pptr;          /* Ptr to previous buff on list */
        struct  rdbuff  *nptr;          /* Ptr to next buffer on list   */
        bool8   found;                  /* Was buff found during search?*/

        /* If device not currently in use, report an error */

        rdptr = &rdstab[devptr->dvminor];
        if (rdptr->rd_state != RD_OPEN) {
                return SYSERR;
        }

        /* If request queue already contains a write request */
        /*    for the block, replace the contents            */

        bptr = rdptr->rd_rhnext;
        while (bptr != (struct rdbuff *)&rdptr->rd_rtnext) {
                if ( (bptr->rd_blknum == blk) &&
                        (bptr->rd_op == RD_OP_WRITE) ) {
                        memcpy(bptr->rd_block, buff, RD_BLKSIZ);
                        return OK;
                }
                bptr = bptr->rd_next;
        }


        /* Search cache for cached copy of block */

        bptr = rdptr->rd_chnext;
        found = FALSE;
```

```
        while (bptr != (struct rdbuff *)&rdptr->rd_ctnext) {
                if (bptr->rd_blknum == blk) {
                        if (bptr->rd_refcnt <= 0) {
                                pptr = bptr->rd_prev;
                                nptr = bptr->rd_next;

                                /* Unlink node from cache list and reset*/
                                /*   the available semaphore accordingly*/

                                pptr->rd_next = bptr->rd_next;
                                nptr->rd_prev = bptr->rd_prev;
                                semreset(rdptr->rd_availsem,
                                        semcount(rdptr->rd_availsem) - 1);
                                found = TRUE;
                        }
                        break;
                }
                bptr = bptr->rd_next;
        }

        if ( !found ) {
                bptr = rdsbufalloc(rdptr);
        }

        /* Create a write request */

        memcpy(bptr->rd_block, buff, RD_BLKSIZ);
        bptr->rd_op = RD_OP_WRITE;
        bptr->rd_refcnt = 0;
        bptr->rd_blknum = blk;
        bptr->rd_status = RD_VALID;
        bptr->rd_pid = getpid();

        /* Insert new request into list just before tail */

        pptr = rdptr->rd_rtprev;
        rdptr->rd_rtprev = bptr;
        bptr->rd_next = pptr->rd_next;
        bptr->rd_prev = pptr;
        pptr->rd_next = bptr;

        /* Signal semaphore to start communication process */

        signal(rdptr->rd_reqsem);
        return OK;
}
```

The code first considers the case where the request queue already contains a pending *write* request for the same block. Note: there can only be one *write* request for a given block on the request queue at a time, which means the queue can be searched in any order. The code searches the queue from the head toward the tail. If it finds a *write* request for the block, *rdswrite* replaces the contents of the request with the new data, and returns.

After searching the request queue, *rdswrite* checks the cache. If the specified block is in the cache, the cached copy must be invalidated. The code searches the cache sequentially. If it finds a match, *rdswrite* removes the buffer from the cache. Instead of moving the buffer to the free list, *rdswrite* uses the buffer to form a request. If no match is found, *rdswrite* calls *rdsbufalloc* to allocate a new buffer for the request.

The final section of *rdswrite* creates a *write* request and inserts it at the tail of the request queue. To help with debugging, the code fills in all fields of the request, even if they are not needed. For example, the process ID field is set to the ID of the calling process, even though the field is not used for a *write* operation.

18.13 The Upper–half Read Function (rdsread)

The second major upper-half function corresponds to the *read* operation. Reading is more complex than writing, because input is synchronous: a process that attempts to read from the disk must be blocked until the data is available. Synchronization of a waiting process uses *send* and *receive*. A node in the request queue contains a process ID field. When a process calls *read*, the driver code creates a *read* request that includes the caller's process ID. It then inserts the request on the request queue, calls *recvclr* to remove pending messages, and calls *receive* to wait for a response. When the request reaches the head of the queue, the remote disk communication process sends a message to the server and receives a response that contains the specified block. The communication process copies the block into the buffer that contains the original request, moves the buffer to the cache, and uses *send* to send a message to the waiting process with the buffer address. The waiting process receives the message, extracts a copy of the data, and returns to the function that called *read*.

As described, the above scheme is insufficient because buffers are used dynamically. To understand the problem, imagine that a low-priority process is blocked waiting to read block 5. Eventually, the communication process obtains block 5 from the server, stores block 5 in the cache, and sends a message to the waiting process. However, assume that while the request is on the request queue, higher-priority application processes begin to execute, meaning that the low-priority process will not run. Unfortunately, if the high-priority processes continue to use disk buffers, the buffer holding block 5 will be taken for another operation.

The problem is exacerbated because the remote disk system permits concurrent access: while one process is waiting to read a block, another process can attempt to read the same block. Thus, when the communication process finally retrieves a copy of the block from the server, multiple processes may need to be informed.

The example code uses a *reference count* technique to handle multiple requests for a block: the header with each buffer contains an integer that counts the number of processes currently reading the block. When a process finishes making a copy, the process decrements the reference count. The code in file *rdsread.c* shows how a process creates a request, enqueues it at the tail of the request list, waits for the request to be filled, and copies data from the request to each caller's buffer; later in the chapter, we will see how the reference count is managed.

```
/* rdsread.c - rdsread */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rdsread  -  Read a block from a remote disk
 *------------------------------------------------------------------------
 */
devcall rdsread (
          struct dentry *devptr,        /* Entry in device switch table */
          char  *buff,                  /* Buffer to hold disk block    */
          int32 blk                     /* Block number of block to read*/
        )
{
        struct  rdscblk *rdptr;         /* Pointer to control block     */
        struct  rdbuff  *bptr;          /* Pointer to buffer possibly   */
                                        /*   in the request list        */
        struct  rdbuff  *nptr;          /* Pointer to "next" node on a  */
                                        /*   list                       */
        struct  rdbuff  *pptr;          /* Pointer to "previous" node   */
                                        /*   on a list                  */
        struct  rdbuff  *cptr;          /* Pointer that walks the cache */

        /* If device not currently in use, report an error */

        rdptr = &rdstab[devptr->dvminor];
        if (rdptr->rd_state != RD_OPEN) {
                return SYSERR;
        }

        /* Search the cache for specified block */

        bptr = rdptr->rd_chnext;
        while (bptr != (struct rdbuff *)&rdptr->rd_ctnext) {
                if (bptr->rd_blknum == blk) {
                        if (bptr->rd_status == RD_INVALID) {
                                break;
                        }
```

```
                memcpy(buff, bptr->rd_block, RD_BLKSIZ);
                return OK;
        }
        bptr = bptr->rd_next;
}

/* Search the request list for most recent occurrence of block */

bptr = rdptr->rd_rtprev;  /* Start at tail of list */

while (bptr != (struct rdbuff *)&rdptr->rd_rhnext) {
    if (bptr->rd_blknum == blk)  {

        /* If most recent request for block is write, copy data */

        if (bptr->rd_op == RD_OP_WRITE) {
                memcpy(buff, bptr->rd_block, RD_BLKSIZ);
                return OK;
        }
        break;
    }
    bptr = bptr->rd_prev;
}

/* Allocate a buffer and add read request to tail of req. queue */

bptr = rdsbufalloc(rdptr);
bptr->rd_op = RD_OP_READ;
bptr->rd_refcnt = 1;
bptr->rd_blknum = blk;
bptr->rd_status = RD_INVALID;
bptr->rd_pid = getpid();

/* Insert new request into list just before tail */

pptr = rdptr->rd_rtprev;
rdptr->rd_rtprev = bptr;
bptr->rd_next = pptr->rd_next;
bptr->rd_prev = pptr;
pptr->rd_next = bptr;

/* Prepare to receive message when read completes */

recvclr();
```

```
        /* Signal semaphore to start communication process */

        signal(rdptr->rd_reqsem);

        /* Block to wait for message */

        bptr = (struct rdbuff *)receive();
        if (bptr == (struct rdbuff *)SYSERR) {
                return SYSERR;
        }
        memcpy(buff, bptr->rd_block, RD_BLKSIZ);
        bptr->rd_refcnt--;
        if (bptr->rd_refcnt <= 0) {

                /* Look for previous item in cache with the same block  */
                /*    number to see if this item was only being kept    */
                /*    until pending read completed                      */

                cptr = rdptr->rd_chnext;
                while (cptr != bptr) {
                        if (cptr->rd_blknum == blk) {

                                /* Unlink from cache */

                                pptr = bptr->rd_prev;
                                nptr = bptr->rd_next;
                                pptr->rd_next = nptr;
                                nptr->rd_prev = pptr;

                                /* Add to the free list */

                                bptr->rd_next = rdptr->rd_free;
                                rdptr->rd_free = bptr;
                        }
                }
        }
        return OK;
}
```

Rdsread begins by handling two special cases. First, if the requested block is found in the cache, *rdsread* extracts a copy of the data and returns. Second, if the request list contains a request to *write* the specified block, *rdsread* extracts a copy of the data from the buffer and returns. Finally, *rdsread* creates a *read* request, enqueues the request at the tail of the request queue, and waits for a message from the communication process as described above.

The code handles one more detail: the case where the reference count reaches zero and a subsequent *read* for the same block has placed a more recent buffer in the cache. If this happens, the more recent version will be used for subsequent *read*s. Therefore, *rdsread* must extract the old buffer from the cache and move it to the free list.

18.14 Flushing Pending Requests

Because *write* does not wait for data transfer, the driver does not inform a process when a *write* operation completes. However, it may be important for the software to know when data is safely stored. For example, an operating system usually ensures that *write* operations are completed before shutdown.

To allow a process to guarantee that all disk transfers have occurred, the driver includes a primitive that will block the calling process until existing requests have been performed. Because "synchronizing" the disk is not a data transfer operation, we use the high-level operation *control*. To flush pending requests, a process calls:

control (disk_device, RD_SYNC)

The driver suspends the calling process until existing requests have been satisfied on the specified device. Once pending operations complete, the call returns.

18.15 The Upper–half Control Function (rdscontrol)

As discussed above, the example driver offers two *control* functions: one to erase an entire disk and one to synchronize data to the disk (i.e., forcing all *write* operations to complete). File *rdscontrol.c* contains the code:

```
/* rdscontrol.c - rdscontrol */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rdscontrol  -  Provide control functions for the remote disk
 *------------------------------------------------------------------------
 */
devcall rdscontrol (
         struct dentry  *devptr,        /* Entry in device switch table */
         int32  func,                   /* The control function to use  */
         int32  arg1,                   /* Argument #1                  */
         int32  arg2                    /* Argument #2                  */
        )
{
```

```
struct  rdscblk *rdptr;         /* Pointer to control block     */
struct  rdbuff  *bptr;          /* Ptr to buffer that will be   */
                                /*   placed on the req. queue   */
struct  rdbuff  *pptr;          /* Ptr to "previous" node on    */
                                /*   a list                     */
struct  rd_msg_dreq msg;        /* Buffer for delete request    */
struct  rd_msg_dres resp;       /* Buffer for delete response   */
char    *to, *from;             /* Used during name copy        */
int32   retval;                 /* Return value                 */

/* Verify that device is currently open */

rdptr = &rdstab[devptr->dvminor];
if (rdptr->rd_state != RD_OPEN) {
        return SYSERR;
}

switch (func) {

/* Synchronize writes */

case RDS_CTL_SYNC:

        /* Allocate a buffer to use for the request list */

        bptr = rdsbufalloc(rdptr);
        if (bptr == (struct rdbuff *)SYSERR) {
                return SYSERR;
        }

        /* Form a sync request */

        bptr->rd_op = RD_OP_SYNC;
        bptr->rd_refcnt = 1;
        bptr->rd_blknum = 0;            /* Unused */
        bptr->rd_status = RD_INVALID;
        bptr->rd_pid = getpid();

        /* Insert new request into list just before tail */

        pptr = rdptr->rd_rtprev;
        rdptr->rd_rtprev = bptr;
        bptr->rd_next = pptr->rd_next;
        bptr->rd_prev = pptr;
        pptr->rd_next = bptr;
```

```
        /* Prepare to wait until item is processed */

        recvclr();
        resume(rdptr->rd_comproc);

        /* Block to wait for message */

        bptr = (struct rdbuff *)receive();
        break;

/* Delete the remote disk (entirely remove it) */

case RDS_CTL_DEL:

        /* Handcraft a message for the server that requests     */
        /*      deleting the disk with the specified ID         */

        msg.rd_type = htons(RD_MSG_DREQ);/* Request deletion    */
        msg.rd_status = htons(0);
        msg.rd_seq = 0; /* rdscomm will insert sequence # later */
        to = msg.rd_id;
        memset(to, NULLCH, RD_IDLEN);   /* Initialize to zeroes */
        from = rdptr->rd_id;
        while ( (*to++ = *from++) != NULLCH ) { /* copy ID      */
                ;
        }

        /* Send message and receive response */

        retval = rdscomm((struct rd_msg_hdr *)&msg,
                                sizeof(struct rd_msg_dreq),
                 (struct rd_msg_hdr *)&resp,
                                sizeof(struct rd_msg_dres),
                                rdptr);

        /* Check response */

        if (retval == SYSERR) {
                return SYSERR;
        } else if (retval == TIMEOUT) {
                kprintf("Timeout during remote file delete\n\r");
                return SYSERR;
        } else if (ntohs(resp.rd_status) != 0) {
                return SYSERR;
        }
```

```
                /* Close local device */

                return rdsclose(devptr);

        default:
                kprintf("rfsControl: function %d not valid\n\r", func);
                return SYSERR;
        }

        return OK;
}
```

The code for each control function should seem familiar. The code to delete an entire disk is similar to the code in *rdsopen* — it creates a message for the server and uses *rdscomm* to send the message. The code to synchronize disk writes is similar to the code in *rdsread* — it creates a request, enqueues the request, calls *recvclr* to remove pending messages, and calls *receive* to wait for a response. Once the response arrives, *rdscontrol* invokes *rdsclose* to close the local device, and returns to its caller.

18.16 Allocating A Disk Buffer (rdsbufalloc)

As we have seen, driver functions call *rdsbufalloc* when they need to allocate a buffer. To understand how *rdsbufalloc* operates, recall that a semaphore counts available buffers either on the free list or in the cache with a reference count of zero. After waiting on the semaphore, *rdsbufalloc* knows that a buffer will exist in one of the two places. It checks the free list first. If the free list is not empty, *rdsbufalloc* extracts the first buffer and returns it. If the free list is empty, *rdsbufalloc* searches the cache for an available buffer, extracts the buffer, and returns it to the caller. If the search completes without finding an available buffer, the count of the semaphore is incorrect, and *rdsbufalloc* calls *panic* to halt the system.

File *rdsbufalloc.c* contains the code:

```
/* rdsbufalloc.c - rdsbufalloc */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rdsbufalloc  -  Allocate a buffer from the free list or the cache
 *------------------------------------------------------------------------
 */
struct rdbuff *rdsbufalloc (
          struct rdscblk *rdptr         /* Ptr to device control block  */
        )
{
        struct  rdbuff  *bptr;          /* Pointer to a buffer          */
        struct  rdbuff  *pptr;          /* Pointer to previous buffer   */
        struct  rdbuff  *nptr;          /* Pointer to next buffer       */

        /* Wait for an available buffer */

        wait(rdptr->rd_availsem);

        /* If free list contains a buffer, extract it */

        bptr = rdptr->rd_free;

        if ( bptr != (struct rdbuff *)NULL ) {
                rdptr->rd_free = bptr->rd_next;
                return bptr;
        }

        /* Extract oldest item in cache that has ref count zero (at     */
        /*   least one such entry must exist because the semaphore      */
        /*   had a nonzero count)                                        */

        bptr = rdptr->rd_ctprev;
        while (bptr != (struct rdbuff *) &rdptr->rd_chnext) {
                if (bptr->rd_refcnt <= 0) {

                        /* Remove from cache and return to caller */

                        pptr = bptr->rd_prev;
                        nptr = bptr->rd_next;
                        pptr->rd_next = nptr;
                        nptr->rd_prev = pptr;
                        return bptr;
                }
```

```
                bptr = bptr->rd_prev;
        }
        panic("Remote disk cannot find an available buffer");
        return (struct rdbuff *)SYSERR;
}
```

18.17 The Upper–half Close Function (rdsclose)

A process invokes *close* to close the remote disk device and stop all communication. To close a remote disk device, all buffers must be moved back to the free list (recreating the conditions immediately following initialization) and the state field in the control block must be assigned *RD_FREE*. Our implementation removes buffers from the cache, but does not handle the request list. Instead, we require a user to wait until all requests have been satisfied and the request list is empty before calling *rdsclose*. The synchronization function, *RDS_CTL_SYNC*,† provides a way to wait for the request queue to drain. File *rdsclose.c* contains the code:

```
/* rdsclose.c - rdsclose */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rdsclose  -  Close a remote disk device
 *------------------------------------------------------------------------
 */
devcall rdsclose (
          struct dentry *devptr         /* Entry in device switch table */
        )
{
        struct  rdscblk *rdptr;         /* Ptr to control block entry   */
        struct  rdbuff  *bptr;          /* Ptr to buffer on a list      */
        struct  rdbuff  *nptr;          /* Ptr to next buff on the list */
        int32   nmoved;                 /* Number of buffers moved      */

        /* Device must be open */

        rdptr = &rdstab[devptr->dvminor];
        if (rdptr->rd_state != RD_OPEN) {
                return SYSERR;
        }

        /* Request queue must be empty */
```

†The synchronization code is found in file *rdscontrol.c* on page 442.

```
        if (rdptr->rd_rhnext != (struct rdbuff *)&rdptr->rd_rtnext) {
                return SYSERR;
        }

        /* Move all buffers from the cache to the free list */

        bptr = rdptr->rd_chnext;
        nmoved = 0;
        while (bptr != (struct rdbuff *)&rdptr->rd_ctnext) {
                nmoved++;

                /* Unlink buffer from cache */

                nptr = bptr->rd_next;
                (bptr->rd_prev)->rd_next = nptr;
                nptr->rd_prev = bptr->rd_prev;

                /* Insert buffer into free list */

                bptr->rd_next = rdptr->rd_free;

                rdptr->rd_free = bptr;
                bptr->rd_status = RD_INVALID;

                /* Move to next buffer in the cache */

                bptr = nptr;
        }

        /* Set the state to indicate the device is closed */

        rdptr->rd_state = RD_FREE;
        return OK;
}
```

18.18 The Lower–half Communication Process (rdsprocess)

In the example implementation, each remote disk device has its own control block, its own set of disk buffers, and its own remote communication process. Thus, a given remote disk process only needs to handle requests from a single queue. Although the code may seem long and filled with details, the general algorithm is straightforward: repeatedly wait on the request semaphore, examine the type of the request at the head of the queue, and either perform a *read*, a *write*, or a *synchronization* operation. File *rdsprocess.c* contains the code:

```
/* rdsprocess.c - rdsprocess */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rdsprocess  -  High-priority background process to repeatedly extract
 *                an item from the request queue and send the request to
 *                the remote disk server
 *------------------------------------------------------------------------
 */
void    rdsprocess (
          struct rdscblk    *rdptr      /* Ptr to device control block  */
        )
{
        struct  rd_msg_wreq msg;        /* Message to be sent           */
                                        /*   (includes data area)       */
        struct  rd_msg_rres resp;       /* Buffer to hold response      */
                                        /*   (includes data area)       */
        int32   retval;                 /* Return value from rdscomm    */
        char    *idto;                  /* Ptr to ID string copy        */
        char    *idfrom;                /* Ptr into ID string           */
        struct  rdbuff  *bptr;          /* Ptr to buffer at the head of */
                                        /*   the request queue          */
        struct  rdbuff  *nptr;          /* Ptr to next buffer on the    */
                                        /*   request queue              */
        struct  rdbuff  *pptr;          /* Ptr to previous buffer       */
        struct  rdbuff  *qptr;          /* Ptr that runs along the      */
                                        /*   request queue              */
        int32   i;                      /* Loop index                   */

        while (TRUE) {                  /* Do forever */

            /* Wait until the request queue contains a node */
            wait(rdptr->rd_reqsem);
            bptr = rdptr->rd_rhnext;

            /* Use operation in request to determine action */

           switch (bptr->rd_op) {

           case RD_OP_READ:

                /* Build a read request message for the server */

                msg.rd_type = htons(RD_MSG_RREQ);       /* Read request */
                msg.rd_status = htons(0);
```

```
msg.rd_seq = 0;         /* Rdscomm fills in an entry    */
idto = msg.rd_id;
memset(idto, NULLCH, RD_IDLEN);/* Initialize ID to zero */
idfrom = rdptr->rd_id;
while ( (*idto++ = *idfrom++) != NULLCH ) { /* Copy ID  */
        ;
}

/* Send the message and receive a response */

retval = rdscomm((struct rd_msg_hdr *)&msg,
                        sizeof(struct rd_msg_rreq),
                 (struct rd_msg_hdr *)&resp,
                        sizeof(struct rd_msg_rres),
                  rdptr );

/* Check response */

if ( (retval == SYSERR) || (retval == TIMEOUT) ||
                (ntohs(resp.rd_status) != 0) ) {
        panic("Failed to contact remote disk server");
}

/* Copy data from the reply into the buffer */

for (i=0; i<RD_BLKSIZ; i++) {
        bptr->rd_block[i] = resp.rd_data[i];
}

/* Unlink buffer from the request queue */

nptr = bptr->rd_next;
pptr = bptr->rd_prev;
nptr->rd_prev = bptr->rd_prev;
pptr->rd_next = bptr->rd_next;

/* Insert buffer in the cache */

pptr = (struct rdbuff *) &rdptr->rd_chnext;
nptr = pptr->rd_next;
bptr->rd_next = nptr;
bptr->rd_prev = pptr;
pptr->rd_next = bptr;
nptr->rd_prev = bptr;
```

```
        /* Initialize reference count */

        bptr->rd_refcnt = 1;

        /* Signal the available semaphore */

        signal(rdptr->rd_availsem);

        /* Send a message to waiting process */

        send(bptr->rd_pid, (uint32)bptr);

        /* If other processes are waiting to read the  */
        /*   block, notify them and remove the request */

        qptr = rdptr->rd_rhnext;
        while (qptr != (struct rdbuff *)&rdptr->rd_rtnext) {
                if (qptr->rd_blknum == bptr->rd_blknum) {
                        bptr->rd_refcnt++;
                        send(qptr->rd_pid,(uint32)bptr);

                        /* Unlink request from queue    */

                        pptr = qptr->rd_prev;
                        nptr = qptr->rd_next;
                        pptr->rd_next = bptr->rd_next;
                        nptr->rd_prev = bptr->rd_prev;

                        /* Move buffer to the free list */

                        qptr->rd_next = rdptr->rd_free;
                        rdptr->rd_free = qptr;
                        signal(rdptr->rd_availsem);
                        break;
                }
                qptr = qptr->rd_next;
        }
        break;

    case RD_OP_WRITE:

        /* Build a write request message for the server */

        msg.rd_type = htons(RD_MSG_WREQ);       /* Write request*/
        msg.rd_blk = bptr->rd_blknum;
```

```
msg.rd_status = htons(0);
msg.rd_seq = 0;         /* Rdscomb fills in an entry    */
idto = msg.rd_id;
memset(idto, NULLCH, RD_IDLEN);/* Initialize ID to zero */
idfrom = rdptr->rd_id;
while ( (*idto++ = *idfrom++) != NULLCH ) { /* Copy ID  */
        ;
}
for (i=0; i<RD_BLKSIZ; i++) {
        msg.rd_data[i] = bptr->rd_block[i];
}

/* Unlink buffer from request queue */

nptr = bptr->rd_next;
pptr = bptr->rd_prev;
pptr->rd_next = nptr;
nptr->rd_prev = pptr;

/* Insert buffer in the cache */

pptr = (struct rdbuff *) &rdptr->rd_chnext;
nptr = pptr->rd_next;
bptr->rd_next = nptr;
bptr->rd_prev = pptr;
pptr->rd_next = bptr;
nptr->rd_prev = bptr;

/* Declare that buffer is eligible for reuse */

bptr->rd_refcnt = 0;
signal(rdptr->rd_availsem);

/* Send the message and receive a response */

retval = rdscomm((struct rd_msg_hdr *)&msg,
                        sizeof(struct rd_msg_wreq),
                  (struct rd_msg_hdr *)&resp,
                        sizeof(struct rd_msg_wres),
                   rdptr );

/* Check response */

if ( (retval == SYSERR) || (retval == TIMEOUT) ||
                 (ntohs(resp.rd_status) != 0) ) {
```

```
                        panic("failed to contact remote disk server");
                }
                break;

            case RD_OP_SYNC:

                /* Send a message to the waiting process */

                send(bptr->rd_pid, OK);

                /* Unlink buffer from the request queue */

                nptr = bptr->rd_next;
                pptr = bptr->rd_prev;
                nptr->rd_prev = bptr->rd_prev;
                pptr->rd_next = bptr->rd_next;

                /* Insert buffer into the free list */

                bptr->rd_next = rdptr->rd_free;
                rdptr->rd_free = bptr;
                signal(rdptr->rd_availsem);
                break;
            }
        }
}
```

When examining the code, remember that the remote disk process has higher priority than any application process. Thus, the code does not need to disable interrupts or use a mutual exclusion semaphore when accessing the request queue, cache, or free list. However, *rdsprocess* must leave all data structures in a valid state before using *rdscomm* to exchange messages with the server, because message reception blocks the calling process (which means other processes can run). In the case of a *read* operation, *rdsprocess* leaves the buffer on the request queue until the request can be satisfied. In the case of a *write* operation, *rdsprocess* extracts a copy of the data and moves the buffer to the cache before calling *rdscomm*.

18.19 Perspective

Conceptually, a remote disk system only needs to provide two basic operations: *read* a block and *write* a block. In practice, however, the issues of synchrony, caching, and sharing dominate the design. Our example simplifies most design decisions because we assume a single Xinu system acts as a client. Thus, the client code manages

its local cache, and does not need to coordinate with other Xinu systems. Similarly, the lack of sharing simplifies the question of synchrony: the client only needs local information to enforce *last-write semantics*.

If the system is extended to permit multiple Xinu systems to share a disk, the entire design must change. A given client cannot cache blocks unless the client coordinates with the server. Furthermore, *last-write semantics* must be enforced across all systems, which means *read* operations need a centralized mechanism to ensure that they occur in order. A tradeoff arises between sharing and efficiency because communication is expensive. Relying on a centralized server to coordinate sharing eliminates caching and imposes a much higher communication overhead. The point is:

> *Extending the remote disk system to include sharing across multiple Xinu systems will result in significant changes to the structure of the system and a significant decrease in performance, especially during periods of simultaneous access.*

18.20 Summary

We considered the design of a remote disk system in which an application can *read* and *write* disk blocks, and the driver uses a network to communicate with a remote server that performs the operation. The driver views a disk as an array of randomly accessible data blocks, and does not provide files, directories, or any index techniques to speed searching. Reading consists of copying data from a specified block on disk into memory; writing consists of copying data from memory onto a specified disk block.

Driver code is divided into upper-half functions that are called by application processes and a lower half that executes as a separate process. Input is synchronous; a process blocks until a request can be satisfied. Output is asynchronous; the driver accepts an outgoing data block, enqueues the request, and returns to the caller immediately without blocking. A process can use the *control* function to flush previous writes to disk.

The driver uses three main data structures: a queue of requests, a cache of recently used blocks, and a free list. Although it relies on caching to make access efficient, our driver guarantees last-write semantics.

EXERCISES

18.1 Redesign the implementation to keep the buffers separate from the nodes used on the request list and the cache (i.e., define nodes for each list and arrange for each node to have a pointer to a buffer). What are the advantages and disadvantages of each approach?

18.2 Redesign the remote disk system to use a "buffer exchange" paradigm in which applications and driver functions share a single buffer pool. To write a disk block, arrange for an application to allocate a buffer, fill the buffer, and pass the buffer when calling *write*. Have *read* return a buffer pointer which the application must free once the data has been extracted.

18.3 It is possible to configure a system with multiple remote disk devices. Modify the code in *rdsopen* to check each open remote disk device to ensure that a disk ID is unique.

18.4 Build a version of a remote disk system that does not use a cache, and measure the difference in performance of the two versions.

18.5 Should requests from high-priority processes take precedence over requests from low-priority processes? Explain why or why not.

18.6 Investigate other algorithms like the "elevator" algorithm that can be used to order disk requests for an electro-mechanical disk.

18.7 Verify that a request to "synchronize" will not return until all pending requests have been satisfied. Is there a bound on the time it can be delayed?

18.8 Redesign the system to use two servers for redundancy. Copy each transaction to both servers. How much overhead does the redundant implementation add?

18.9 Build a remote disk server that allows simultaneous access by multiple client computers and multiple concurrent processes on each client. Plot performance as the number of simultaneous users increases.

18.10 Some operating systems allow a disk to be divided into multiple *partitions*, where each partition has blocks numbered 0 through $N-1$ for some N. What is the advantage of partitioning disks? Hint: consider the previous exercise.

Chapter Contents

19.1 What Is A File System?, 459
19.2 An Example Set Of File Operations, 460
19.3 Design Of A Local File System, 461
19.4 Data Structures For The Xinu File System, 461
19.5 Implementation Of The Index Manager, 462
19.6 Clearing An Index Block (lfibclear), 467
19.7 Retrieving An Index Block (lfibget), 468
19.8 Storing An Index Block (lfibput), 469
19.9 Allocating An Index Block From The Free List (lfiballoc), 471
19.10 Allocating A Data Block From The Free List (lfdballoc), 472
19.11 Using The Device-Independent I/O Functions For Files, 474
19.12 File System Device Configuration And Function Names, 474
19.13 The Local File System Open Function (lfsopen), 475
19.14 Closing A File Pseudo-Device (lflclose), 483
19.15 Flushing Data To Disk (lfflush), 483
19.16 Bulk Transfer Functions For A File (lflwrite, lflread), 486
19.17 Seeking To A New Position In the File (lflseek), 488
19.18 Extracting One Byte From A File (lflgetc), 489
19.19 Changing One Byte In A File (lflputc), 490
19.20 Loading An Index Block And A Data Block (lfsetup), 492
19.21 Master File System Device Initialization (lfsinit), 496
19.22 Pseudo-Device Initialization (lflinit), 497
19.23 File Truncation (lftruncate), 499
19.24 Initial File System Creation (lfscreate), 501
19.25 Perspective, 503
19.26 Summary, 504

19

File Systems

Filing is concerned with the past; anything you actually need to see again has to do with the future.

— Katharine Whitehorn

The previous chapter discusses a disk abstraction, and describes a hardware interface that allows the system to read and write individual blocks. Although disks have the advantage of providing long-term, non-volatile storage, the block-oriented interface is cumbersome.

This chapter introduces the file system abstraction. It shows how an operating system manages a set of dynamically changing file objects, and how the system maps files onto the underlying disk hardware.

19.1 What Is A File System?

A *file system* consists of software that manages permanent data objects whose values persist longer than the processes that create and use them. Permanent data is kept in *files*, which are stored on secondary storage devices, either solid state or electromechanical disks. Files are organized into *directories* (also called *folders*). Conceptually, each file consists of a sequence of data objects (e.g., a sequence of integers). The file system provides operations that *create* or *delete* a file, *open* a file given its name, *read* the next object from an open file, *write* an object onto an open file, or *close* a file. If a file system allows random access, the file interface also provides a way a process can *seek* to a specified location in a file.

Many file systems offer more functionality than an interface that can access individual files on secondary storage — they provide an abstract namespace and high-level operations to manipulate objects in that space. The file namespace consists of the set of valid file names. A namespace can be as simple as "the set of strings formed from at least one but fewer than nine alphabetic characters," or as complex as "the set of strings that form a valid encoding of the network, machine, user, subdirectory, and file identifiers in a specified syntax." In some systems, the syntax of names in the abstract space conveys information about their type (e.g., text files end in ".txt"). In others, names give information about the organization of the file system (e.g., a file name that begins with the string "M1_d0:" might reside on disk 0 of machine 1). We will defer a discussion of file naming to Chapter 21, and concentrate on file access.

19.2 An Example Set Of File Operations

Our example system uses a straightforward approach motivated by a desire to unify the interface between devices and files and to keep the file system software small. File semantics are taken from Unix according to the following principle:

> *The file system considers each file to be a sequence of zero or more bytes; any further structure must be enforced by application programs that use the file.*

Treating a file as a sequence of bytes has several advantages. First, the file system does not impose a type on the file and does not need to distinguish among file types. Second, the code is small because a single set of file system functions suffices for all files. Third, the file semantics can be applied to devices and services as well as to conventional files. Fourth, application programs can choose an arbitrary structure for data without changing the underlying system. Finally, file contents are independent of the processor or memory (e.g., an application may need to distinguish among a 32-bit and 64-bit integer stored in a file, but the file system does not).

Our system will use exactly the same high-level operations for files that are used for devices. Thus, the file system will support *open*, *close*, *read*, *write*, *putc*, *getc*, *seek*, *init*, and *control* functions. When applied to conventional files, the operations produce the following results. *Init* initializes data structures at startup. *Open*ing a named file connects an executing process with the data on disk, and establishes a pointer to the first byte. Operations *getc* and *read* retrieve data from the file and advance the pointer; *getc* retrieves one byte, and *read* can retrieve multiple bytes. Operations *putc* and *write* change bytes in the file and move the pointer along, extending the file length if new data is written beyond the end; *putc* changes one byte, and *write* can change multiple bytes. The *seek* operation moves the pointer to a specified byte position in the file; the first byte is at position zero. Finally, *close* detaches the running process from the file, leaving the data in the file on permanent storage.

19.3 Design Of A Local File System

A file is said to be *local* to a given computer if the file resides on a storage device that is attached to the computer. The design of software that manages such files is non-trivial; it has been the subject of much research. Although the file operations may seem straightforward, complexity arises because files are *dynamic*. That is, a single disk can hold multiple files, and a given file can grow arbitrarily large (until disk space is exhausted). To permit dynamic file growth, a system cannot pre-allocate disk blocks for a file. Thus, dynamic data structures are needed.

A second form of complexity arises from concurrency. To what extent should the system support concurrent file operations? Large systems usually allow arbitrary numbers of processes to read and write arbitrary numbers of files concurrently. The chief difficulty with multiple access lies in specifying exactly what it means to have multiple processes writing and reading a file at the same time. When will data become available for reading? If two processes attempt to write to the same data byte in the file, which will be accepted? Can a process lock pieces of a file to avoid interference?

The generality of allowing multiple processes to read and write a file is usually not necessary on small embedded systems. Thus, to limit the software complexity and make better use of disk space, small systems can constrain the ways in which files can be accessed. They may limit the number of files that a given process can access simultaneously, or limit the number of processes that can access a given file simultaneously.

Our goal is to design efficient, compact file system software that allows processes to create and extend files dynamically without incurring unnecessary overhead. As a compromise between generality and efficiency, we will allow a process to open an arbitrary number of files until resources are exhausted. However, the system limits a file to one active open. That is, if a file is open, successive requests to open the file (e.g., requests by other processes) will fail until the file has been closed. Each file has a mutual exclusion semaphore to guarantee that only one process at a time can attempt to write a byte to the file, read a byte from the file, or change the current file position. Furthermore, the directory has a mutual exclusion semaphore to guarantee that only one process at a time can attempt to create a file or otherwise change a directory entry. Although concurrency requires attention to detail, the most significant consequence of our design arises from its support for dynamic file growth: data structures will be needed to allocate space on a disk dynamically. The next section explains the data structures used.

19.4 Data Structures For The Xinu File System

To support dynamic growth and random access, the Xinu file system allocates disk blocks dynamically and uses an *index mechanism* to locate the data in a given file quickly. The design partitions a disk into three separate areas as Figure 19.1 illustrates: a *directory*, an *index area*, and a *data area*.

Figure 19.1 Illustration of a disk partitioned into three areas for the Xinu file system.

The first sector of the disk holds a directory that contains a list of file names along with a pointer to the list of index blocks for the file. The directory also contains two other pointers: one to a list of free (unused) index blocks and another to a list of free data blocks. The directory entry for a file also contains an integer that gives the current size of the file measured in bytes.

Following the directory, the disk contains an index area that holds a set of *index blocks*, abbreviated *i-blocks*. Each file has its own index, which consists of a singly-linked list of index blocks. Initially, all index blocks are linked onto a free list from which the system allocates one as needed; index blocks are only returned to the free list if a file is truncated or deleted.

Following the index area, remaining blocks of the disk comprise a data area. Each block in the data area is referred to as a *data block*, abbreviated *d-block*, because a block contains data that has been stored in a file. Once a data block has been allocated to a file, the block only can contain data. A data block does not contain pointers to other data blocks, nor does it contain information that relates the block to the file of which it is a part; all such information resides in the file's index.

Similar to index blocks, when a disk is initialized, the data blocks are linked onto a free list. The file system allocates data blocks from the free list as needed, and returns data blocks to the free list when a file is truncated or deleted.

Figure 19.2 illustrates the conceptual data structure used for a Xinu file system. The figure is not drawn to scale: in practice a data block is much larger than an index block and occupies one physical disk block. The important idea is that the data structure illustrated in the figure resides on disk. We will see that at any given time, only a few pieces of the structure are present in memory — the file system must create and maintain an index without reading the structure into memory.

19.5 Implementation Of The Index Manager

Conceptually index blocks form a randomly accessible array that is mapped onto a contiguous area of the disk. Thus, index blocks are numbered from zero through K, and the software uses the number to refer to a given index block. Because an index block is smaller than a physical disk block, the system stores seven index blocks into each physical block, and the software handles the details of reading and writing an individual index block.

Figure 19.2 Illustration of the Xinu file system, where each file consists of a linked list of index blocks that each contain pointers to data blocks.

Because the underlying hardware can only transfer an entire disk block at a time, the file system cannot transfer an individual index block without transferring others that reside in the same physical disk block. Therefore, to write an index block, the software must read the entire physical disk block in which it resides, copy the new index block into the correct position, and write the resulting physical block back to disk. Similarly, to read an index block, the software must read the physical disk block in which it resides, and then extract the index block.

Before we examine the code to handle index blocks, we need to understand basic definitions. File *lfilesys.h* defines constants and data structures used throughout the local file system, including struct *lfiblk* which defines the contents of an index block. As the file shows, each index block contains a pointer to the next index block, an *offset* that specifies the lowest position in the file indexed by the block, and an array of sixteen pointers to data blocks. That is, each entry in the array gives the physical disk sector number of a data block. Because a sector is 512 bytes long, a single index block indexes sixteen 512-byte blocks or 8192 bytes of data.

How does the software know where to find an index block given its address? Index blocks are contiguous, and occupy contiguous disk sectors starting at sector *LF_AREA_IB*. In our design, the directory occupies disk block zero, which means that the index area starts at sector one. Thus, index blocks zero through six lie in sector one, seven through thirteen lie in sector two, and so on. Inline function *ib2sect* converts an index block number into the correct sector number, and inline function *ib2disp* converts an index block number to a byte displacement within a physical disk block. Both functions can be found in file *lfilesys.h*.

```
/* lfilesys.h - ib2sect, ib2disp */

/************************************************************************/
/*                                                                      */
/*               Local File System Data Structures                      */
/*                                                                      */
/*   A local file system uses a random-access disk composed of 512-byte */
/* sectors numbered 0 through N-1.  We assume disk hardware can read or */
/* write any sector at random, but must transfer an entire sector.      */
/* Thus, to write a few bytes, the file system must read the sector,    */
/* replace the bytes, and then write the sector back to disk.  Xinu's   */
/* local file system divides the disk as follows: sector 0 is a         */
/* directory, the next K sectors constitute an index area, and the      */
/* remaining sectors comprise a data area. The data area is easiest to  */
/* understand: each sector holds one data block (d-block) that stores   */
/* contents from one of the files (or is on a free list of unused data  */
/* blocks).  We think of the index area as holding an array of index    */
/* blocks (i-blocks) numbered 0 through I-1.  A given sector in the     */
/* index area holds 7 of the index blocks, which are each 72 bytes      */
/* long.  Given an i-block number, the file system must calculate the   */
/* disk sector in which the i-block is located and the byte offset      */
/* within the sector at which the i-block resides.  Internally, a file  */
/* is known by the i-block index of the first i-block for the file.     */
/* The directory contains a list of file names and the  i-block number  */
/* of the first i-block for the file.  The directory also holds the     */
/* i-block number for a list of free i-blocks and a data block number   */
/* of the first data block on a list of free data blocks.               */
/*                                                                      */
/************************************************************************/

#ifndef Nlfl
#define Nlfl    1
#endif

/* Use the remote disk device if no disk is defined (file system  */
/*  *assumes* the underlying disk has a block size of 512 bytes)  */

#ifndef LF_DISK_DEV
#define LF_DISK_DEV     SYSERR
#endif

#define LF_MODE_R       F_MODE_R        /* Mode bit for "read"          */
#define LF_MODE_W       F_MODE_W        /* Mode bit for "write"         */
#define LF_MODE_RW      F_MODE_RW       /* Mode bits for "read or write"*/
```

```
#define LF_MODE_O       F_MODE_O        /* Mode bit for "old"           */
#define LF_MODE_N       F_MODE_N        /* Mode bit for "new"           */

#define LF_BLKSIZ       512             /* Assumes 512-byte disk blocks */
#define LF_NAME_LEN     16              /* Length of name plus null     */
#define LF_NUM_DIR_ENT  20              /* Num. of files in a directory */

#define LF_FREE         0               /* Slave device is available    */
#define LF_USED         1               /* Slave device is in use       */

#define LF_INULL        (ibid32) -1     /* Index block null pointer     */
#define LF_DNULL        (dbid32) -1     /* Data block null pointer      */
#define LF_IBLEN        16              /* Data block ptrs per i-block  */
#define LF_IDATA        8192            /* Bytes of data indexed by a   */
                                        /*   single index block         */
#define LF_IMASK        0x00001fff      /* Mask for the data indexed by */
                                        /*   one index block (i.e.,     */
                                        /*   bytes 0 through 8191).     */
#define LF_DMASK        0x000001ff      /* Mask for the data in a data  */
                                        /*   block (0 through 511)      */

#define LF_AREA_IB      1               /* First sector of i-blocks     */
#define LF_AREA_DIR     0               /* First sector of directory    */

/* Structure of an index block on disk */

struct  lfiblk          {               /* Format of index block        */
        ibid32          ib_next;        /* Address of next index block  */
        uint32          ib_offset;      /* First data byte of the file  */
                                        /*  Indexed by this i-block     */
        dbid32          ib_dba[LF_IBLEN];/* Ptrs to data blocks indexed */
};

/* Conversion functions below assume 7 index blocks per disk block */

/* Conversion between index block number and disk sector number */

#define ib2sect(ib)     (((ib)/7)+LF_AREA_IB)

/* Conversion between index block number and the relative offset within */
/*      a disk sector                                                   */

#define ib2disp(ib)     (((ib)%7)*sizeof(struct lfiblk))
```

```
/* Structure used in each directory entry for the local file system */

struct  ldentry {                       /* Description of entry for one */
                                        /*   file in the directory      */
        uint32  ld_size;                /* Curr. size of file in bytes  */
        ibid32  ld_ilist;               /* ID of first i-block for file */
                                        /*   or IB_NULL for empty file  */
        char    ld_name[LF_NAME_LEN];   /* Null-terminated file name    */
};

/* Structure of a data block when on the free list on disk */

struct  lfdbfree {
        dbid32  lf_nextdb;              /* Next data block on the list  */
        char    lf_unused[LF_BLKSIZ - sizeof(dbid32)];
};

/* Format of the file system directory, either on disk or in memory */

#pragma pack(2)
struct  lfdir   {                       /* Entire directory on disk     */
        dbid32  lfd_dfree;              /* List of free d-blocks on disk*/
        ibid32  lfd_ifree;              /* List of free i-blocks on disk*/
        int32   lfd_nfiles;             /* Current number of files      */
        struct  ldentry lfd_files[LF_NUM_DIR_ENT]; /* Set of files      */
        char    padding[20];            /* Unused chars in directory blk*/
};
#pragma pack()

/* Global data used by local file system */

struct  lfdata  {                       /* Local file system data       */
        did32   lf_dskdev;              /* Device ID of disk to use     */
        sid32   lf_mutex;               /* Mutex for the directory and  */
                                        /*   index/data free lists      */
        struct  lfdir   lf_dir;         /* In-memory copy of directory  */
        bool8   lf_dirpresent;          /* True when directory is in    */
                                        /*   memory (1st file is open)  */
        bool8   lf_dirdirty;            /* Has the directory changed?   */
};

/* Control block for local file pseudo-device */

struct  lflcblk {                       /* Local file control block     */
                                        /*   (one for each open file)   */
```

```
        byte    lfstate;                /* Is entry free or used        */
        did32   lfdev;                  /* Device ID of this device     */
        sid32   lfmutex;                /* Mutex for this file          */
        struct  ldentry *lfdirptr;      /* Ptr to file's entry in the   */
                                        /*   in-memory directory        */
        int32   lfmode;                 /* Mode (read/write/both)       */
        uint32  lfpos;                  /* Byte position of next byte   */
                                        /*   to read or write           */
        char    lfname[LF_NAME_LEN];    /* Name of the file             */
        ibid32  lfinum;                 /* ID of current index block in */
                                        /*   lfiblock or LF_INULL       */
        struct  lfiblk  lfiblock;       /* In-mem copy of current index */
                                        /*   block                      */
        dbid32  lfdnum;                 /* Number of current data block */
                                        /*   in lfdblock or LF_DNULL    */
        char    lfdblock[LF_BLKSIZ];    /* In-mem copy of current data  */
                                        /*   block                      */
        char    *lfbyte;                /* Ptr to byte in lfdblock or   */
                                        /*   address one beyond lfdblock*/
                                        /*   if current file pos lies   */
                                        /*   outside lfdblock           */
        bool8   lfibdirty;              /* Has lfiblock changed?        */
        bool8   lfdbdirty;              /* Has lfdblock changed?        */
};

extern  struct  lfdata  Lf_data;
extern  struct  lflcblk lfltab[];

/* Control functions */

#define LF_CTL_DEL      F_CTL_DEL       /* Delete a file                */
#define LF_CTL_TRUNC    F_CTL_TRUNC     /* Truncate a file              */
#define LF_CTL_SIZE     F_CTL_SIZE      /* Obtain the size of a file    */
```

19.6 Clearing An Index Block (lfibclear)

Whenever it allocates an index block from the free list, the file system must read the index block into memory and clear the block to remove old information. In particular, all data block pointers must be set to a null value, so they will not be confused with valid pointers. Furthermore, the offset in the index block must be assigned the appropriate offset in the file. Function *lfibclear* clears an index block; file *lfibclear.c* contains the code.

```
/* lfibclear.c - lfibclear */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  lfibclear  --  Clear an in-core copy of an index block
 *------------------------------------------------------------------------
 */
void    lfibclear(
          struct lfiblk *ibptr,         /* Address of i-block in memory */
          int32         offset          /* File offset for this i-block */
        )
{
        int32   i;                      /* Index for data block array   */

        ibptr->ib_offset = offset;      /* Assign specified file offset */
        for (i=0 ; i<LF_IBLEN ; i++) {  /* Clear each data block pointer*/
                ibptr->ib_dba[i] = LF_DNULL;
        }
        ibptr->ib_next = LF_INULL;      /* Set next ptr to null         */
        return;
}
```

19.7 Retrieving An Index Block (lfibget)

To read an index block into memory, the system must map the index block number to a physical disk block address, read the physical disk block, and copy the appropriate area from the physical block into the specified memory location. File *lfibget.c* contains the code, which uses inline function *ib2sect* to convert the index block number to a disk sector, and function *ib2disp* to compute the location of the index block within the disk sector.

```
/* lfibget.c - lfibget */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  lfibget  -  Get an index block from disk given its number (assumes
 *                      mutex is held)
 *------------------------------------------------------------------------
 */
void    lfibget(
          did32         diskdev,        /* Device ID of disk to use     */
```

```
          ibid32        inum,           /* ID of index block to fetch   */
          struct lfiblk *ibuff          /* Buffer to hold index block   */
        )
{
        char    *from, *to;             /* Pointers used in copying     */
        int32   i;                      /* Loop index used during copy  */
        char    dbuff[LF_BLKSIZ];       /* Buffer to hold disk block    */

        /* Read disk block that contains the specified index block */

        read(diskdev, dbuff, ib2sect(inum));

        /* Copy specified index block to caller's ibuff */

        from = dbuff + ib2disp(inum);
        to = (char *)ibuff;
        for (i=0 ; i<sizeof(struct lfiblk) ; i++)
                *to++ = *from++;
        return;
}
```

19.8 Storing An Index Block (lfibput)

Conceptually, we think of index blocks as occupying a giant array on disk. However, changing an index block is much more complex than changing an element in an array for two reasons. First, because the file system does not keep a record of which items in an index block changed, the entire index block must be rewritten. Second, the disk abstraction only provides write capability for an entire sector at a time. However, an index block only occupies part of a sector.

To write an index block without changing other items in the sector, the file system must read an entire disk sector from disk, copy bytes of the index block into the appropriate area, and then write the entire sector back to disk. File *lfibput.c* contains the code. *Lfibput* is given a disk device, an index block ID, and a buffer address as arguments. The code uses the same inline functions as *lfibget* to convert the index block ID to a disk sector and to find the byte displacement in the sector for the specified index block. It then calls *read* to read the appropriate disk block. It copies the index block from the caller's buffer into the disk block, and calls *write* to write the entire disk block back to the disk.

```
/* lfibput.c - lfibput */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  lfibput  -  Write an index block to disk given its ID (assumes
 *                      mutex is held)
 *------------------------------------------------------------------------
 */
status  lfibput(
          did32         diskdev,        /* ID of disk device            */
          ibid32        inum,           /* ID of index block to write   */
          struct lfiblk *ibuff          /* Buffer holding the index blk */
        )
{
        dbid32  diskblock;              /* ID of disk sector (block)    */
        char    *from, *to;             /* Pointers used in copying     */
        int32   i;                      /* Loop index used during copy  */
        char    dbuff[LF_BLKSIZ];       /* Temp. buffer to hold d-block */

        /* Compute disk block number and offset of index block */

        diskblock = ib2sect(inum);
        to = dbuff + ib2disp(inum);
        from = (char *)ibuff;

        /* Read disk block */

        if (read(diskdev, dbuff, diskblock) == SYSERR) {
                return SYSERR;
        }

        /* Copy index block into place */

        for (i=0 ; i<sizeof(struct lfiblk) ; i++) {
                *to++ = *from++;
        }

        /* Write the block back to disk */

        write(diskdev, dbuff, diskblock);
        return OK;
}
```

19.9 Allocating An Index Block From The Free List (lfiballoc)

The file system allocates an index block from the free list whenever it needs to extend the index for a file. Function *lfiballoc* obtains the next free index block for the disk, and returns its identifier. The code, which is found in file *lfiballoc.c*, assumes that a copy of the directory for the file system has been read into memory and placed in global variable *Lf_data.lf_dir*. Once the index block has been unlinked from the free list, the directory is written back to disk.

```
/* lfiballoc.c - lfiballoc */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lfiballoc  -  Allocate a new index block from free list on disk
 *                      (assumes directory mutex held)
 *------------------------------------------------------------------------
 */
ibid32  lfiballoc (void)
{
        ibid32  ibnum;          /* ID of next block on the free list    */
        struct  lfiblk  iblock; /* Buffer to hold an index block        */

        /* Get ID of first index block on free list */

        ibnum = Lf_data.lf_dir.lfd_ifree;
        if (ibnum == LF_INULL) {        /* Ran out of free index blocks */
                panic("out of index blocks");
        }
        lfibget(Lf_data.lf_dskdev, ibnum, &iblock);

        /* Unlink index block from the directory free list */

        Lf_data.lf_dir.lfd_ifree = iblock.ib_next;

        /* Write a copy of the directory to disk after the change */

        write(Lf_data.lf_dskdev, (char *) &Lf_data.lf_dir, LF_AREA_DIR);
        Lf_data.lf_dirdirty = FALSE;

        return ibnum;
}
```

19.10 Allocating A Data Block From The Free List (lfdballoc)

Because an index block contains a "next" pointer field, linking them into a free list is straightforward. For data blocks, however, the free list is less obvious because a data block does not usually contain a pointer field. The Xinu design uses a singly-linked free list, which means that only one pointer is needed. When a data block is on the free list, the system uses the first four bytes of the data block to store a pointer to the next block on the list. Structure *lfdbfree*, found in file *lfilesys.h* above, defines the format of a block on the free list. Whenever it extracts a data block from the free list, the file system uses the structure definition. Of course, once a block has been removed from the free list and allocated to a file, the block is treated as an array of bytes.

Function *lfdballoc*, which allocates a data block from the free list and returns the block number, illustrates how the system uses struct *lfdbfree*. The code can be found in file *lfdballoc.c*.

```
/* lfdballoc.c - lfdballoc */

#include <xinu.h>

#define  DFILL  '+'             /* character used to fill a disk block  */

/*------------------------------------------------------------------------
 * lfdballoc  -  Allocate a new data block from free list on disk
 *                      (assumes directory mutex held)
 *------------------------------------------------------------------------
 */
dbid32  lfdballoc (
          struct lfdbfree *dbuff /* Addr. of buffer to hold data block  */
        )
{
        dbid32  dnum;           /* ID of next d-block on the free list  */
        int32   retval;         /* Return value                         */

        /* Get the ID of first data block on the free list */

        dnum = Lf_data.lf_dir.lfd_dfree;
        if (dnum == LF_DNULL) { /* Ran out of free data blocks */
                panic("out of data blocks");
        }
        retval = read(Lf_data.lf_dskdev, (char *)dbuff, dnum);
        if (retval == SYSERR) {
                panic("lfdballoc cannot read disk block\n\r");
        }
```

```
        /* Unlink d-block from in-memory directory */

        Lf_data.lf_dir.lfd_dfree = dbuff->lf_nextdb;
        write(Lf_data.lf_dskdev, (char *)&Lf_data.lf_dir, LF_AREA_DIR);
        Lf_data.lf_dirdirty = FALSE;

        /* Fill data block to erase old data */

        memset((char *)dbuff, DFILL, LF_BLKSIZ);
        return dnum;
}
```

A corresponding function, *lfdbfree*, found in file *lfdbfree.c*, returns a block to the free list. The code inserts a pointer into the first four bytes of a block, pointing to the current free list. It then makes the free list pointer in the directory point to the block. Both the data block and the directory must be written to disk after they have been changed.

```
/* lfdbfree.c - lfdbfree */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  lfdbfree  -  Free a data block given its block number (assumes
 *                      directory mutex is held)
 *------------------------------------------------------------------------
 */
status  lfdbfree(
          did32         diskdev,        /* ID of disk device to use     */
          dbid32        dnum            /* ID of data block to free     */
        )
{
        struct  lfdir   *dirptr;        /* Pointer to directory         */
        struct  lfdbfree buf;           /* Buffer to hold data block    */

        dirptr = &Lf_data.lf_dir;
        buf.lf_nextdb = dirptr->lfd_dfree;
        dirptr->lfd_dfree = dnum;
        write(diskdev, (char *)&buf,   dnum);
        write(diskdev, (char *)dirptr, LF_AREA_DIR);

        return OK;
}
```

19.11 Using The Device-Independent I/O Functions For Files

The file system software must establish connections between running processes and disk files to allow operations like *read* and *write* to be mapped onto the correct file. Exactly how the system performs this mapping depends on both the size and generality needed. To keep our system small, we avoid introducing new functions by reusing the device switch mechanisms that are already in place.

Imagine that a set of file *pseudo-devices* has been added to the device switch table such that each pseudo-device can be used to control an open file. As with conventional devices, a pseudo-device has a set of driver functions that perform *read*, *write*, *getc*, *putc*, *seek*, and *close* operations. When a process opens a disk file, the file system searches for a currently unused pseudo-device, sets up the control block for the pseudo-device, and returns the ID of the pseudo-device to the caller. After the file has been opened, the process uses the device ID with operations *getc*, *read*, *putc*, *write*, and *seek*. The device switch table maps each high-level operation to the appropriate driver function for file pseudo-devices exactly as it maps high-level operations onto device drivers for physical devices. Finally, when it finishes using a file, a process calls *close* to break the connection and make the pseudo-device available for use with another file. The details will become clear as we review the code.

Designing a pseudo-device driver is not unlike designing a device driver for a conventional hardware device. Just like other drivers, the pseudo-device driver creates a control block for each pseudo-device. The control block for a file pseudo-device uses struct *lflcblk*, which is defined in file *lfilesys.h*.† Conceptually, the control block contains two types of items: fields that hold information about the pseudo-device and fields that hold information from the disk. Fields *lfstate* and *lfmode* are the former type: the state field specifies whether the device is currently in use, and the mode field specifies whether the file has been opened for reading, writing, or both. Fields *lfiblock* and *lfdblock* are of the latter type: when a file is being read or written they contain a copy of the index block and the data block for the current position in the file measured in bytes (which is given by field *lfpos*).

When a file is opened, the position (field *lfpos* in the control block) is assigned zero. As processes *read* or *write* data, the position increases. A process can call *seek* to move to an arbitrary position in the file, and *lfpos* is updated.

19.12 File System Device Configuration And Function Names

What interface should be used to open a file and allocate a pseudo-device for reading and writing? One possible design adds a new file abstraction to the operating system. For, example, we can imagine a function *fileopen* that takes three arguments (a disk device, file name, and mode) and returns the descriptor of a pseudo-device for the file:

```
fd = fileopen(device, filename, mode);
```

†File *lfilesys.h* can be found on page 464.

In contrast with Unix, Xinu takes a device approach:

> In Unix, everything is a file; in Xinu, everything is a device.

That is, Xinu tends to map many functions into the device space. The local file system follows the pattern by defining a *master* local file device, *LFILESYS*. Calling *open* on the master device causes the system to allocate a pseudo-device and return the device ID of the pseudo-device. File pseudo-devices are named *LFILE0*, *LFILE1*, ..., but the names are used only in the configuration file — a process receives the device descriptor of a pseudo-device when it opens the master device, but no process should call *open* on a pseudo-device. Figure 19.3 shows the types used to configure both the master and the local file pseudo-devices.

```
/* Local File System master device type */

lfs: on disk
                -i lfsinit      -o lfsopen      -c ioerr
                -r ioerr        -g ioerr        -p ioerr
                -w ioerr        -s ioerr        -n rfscontrol
                -intr NULL

/* Local file pseudo-device type */

lfl: on lfs
                -i lflinit      -o ioerr        -c lflclose
                -r lflread      -g lflgetc      -p lflputc
                -w lflwrite     -s lflseek      -n ioerr
                -intr NULL
```

Figure 19.3 Configuration types for a local file system master device and local file pseudo-devices.

As the figure shows, driver functions for the master file system device have names that begin with *lfs*, and driver functions for a file pseudo-device have names that begin with *lfl*. We will see that support functions used by either set of driver functions have names that begin with *lf*.

19.13 The Local File System Open Function (lfsopen)

Figure 19.4 shows the configuration of the local file system master device and a set of seven local file pseudo-devices. Because a pseudo-device is used for each open file, the number of local file pseudo-devices provides a bound on the number of files that can be opened simultaneously.

```
/* Local file system master device (one per system)  */

            LFILESYS is lfs on disk

/* Local file pseudo-devices (many per system)       */

            LFILE0 is lfl on lfs
            LFILE1 is lfl on lfs
            LFILE2 is lfl on lfs
            LFILE3 is lfl on lfs
            LFILE4 is lfl on lfs
            LFILE5 is lfl on lfs
            LFILE6 is lfl on lfs
```

Figure 19.4 Configuration of a local file system master device and a set of local file pseudo-devices.

A process uses the master device to open a local file, and then uses the pseudo-device to access the file. For example, to open a local file named *myfile* for reading and writing, a programmer codes:

```
fd = open(LFILESYS, "myfile", "rw");
```

Assuming the open succeeds, descriptor *fd* can be used to write data to the file, as in the following:

```
char buffer[1500];
... code to fill buffer ...
fd = write(fd, buffer, 1500);
```

Device *LFILESYS* is only used to open a file. Therefore, the master file system device driver only needs functions for *open* and *init*; all other I/O operations map to *ioerr*. Function *lfsopen* performs the *open* operation; the code can be found in file *lfsopen.c*.

```
/* lfsopen.c - lfsopen */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lfsopen - Open a file and allocate a local file pseudo-device
 *------------------------------------------------------------------------
 */
devcall lfsopen (
         struct dentry  *devptr,        /* Entry in device switch table */
         char   *name,                  /* Name of file to open         */
         char   *mode                   /* Mode chars: 'r' 'w' 'o' 'n'  */
        )
```

```
{
        struct  lfdir   *dirptr;        /* Ptr to in-memory directory   */
        char            *from, *to;     /* Ptrs used during copy        */
        char            *nam, *cmp;     /* Ptrs used during comparison  */
        int32           i;              /* General loop index           */
        did32           lfnext;         /* Minor number of an unused    */
                                        /*    file pseudo-device        */
        struct  ldentry *ldptr;         /* Ptr to an entry in directory */
        struct  lflcblk *lfptr;         /* Ptr to open file table entry */
        bool8           found;          /* Was the name found?          */
        int32   retval;                 /* Value returned from function */
        int32   mbits;                  /* Mode bits                    */

        /* Check length of name file (leaving space for NULLCH */

        from = name;
        for (i=0; i< LF_NAME_LEN; i++) {
                if (*from++ == NULLCH) {
                        break;
                }
        }
        if (i >= LF_NAME_LEN) {         /* Name is too long */
                return SYSERR;
        }

        /* Parse mode argument and convert to binary */

        mbits = lfgetmode(mode);
        if (mbits == SYSERR) {
                return SYSERR;
        }

        /* If named file is already open, return SYSERR */

        lfnext = SYSERR;
        for (i=0; i<Nlfl; i++) {        /* Search file pseudo-devices   */
                lfptr = &lfltab[i];
                if (lfptr->lfstate == LF_FREE) {
                        if (lfnext == SYSERR) {
                                lfnext = i; /* Record index */
                        }
                        continue;
                }
```

```
            /* Compare requested name to name of open file */

            nam = name;
            cmp = lfptr->lfname;
            while(*nam != NULLCH) {
                    if (*nam != *cmp) {
                            break;
                    }
                    nam++;
                    cmp++;
            }

            /* See if comparison succeeded */

            if ( (*nam==NULLCH) && (*cmp == NULLCH) ) {
                    return SYSERR;
            }
    }
    if (lfnext == SYSERR) { /* No slave file devices are available  */
            return SYSERR;
    }

    /* Obtain copy of directory if not already present in memory    */

    dirptr = &Lf_data.lf_dir;
    wait(Lf_data.lf_mutex);
    if (! Lf_data.lf_dirpresent) {
        retval = read(Lf_data.lf_dskdev,(char *)dirptr,LF_AREA_DIR);
        if (retval == SYSERR ) {
            signal(Lf_data.lf_mutex);
            return SYSERR;
        }
        Lf_data.lf_dirpresent = TRUE;
    }

    /* Search directory to see if file exists */

    found = FALSE;
    for (i=0; i<dirptr->lfd_nfiles; i++) {
            ldptr = &dirptr->lfd_files[i];
            nam = name;
            cmp = ldptr->ld_name;
            while(*nam != NULLCH) {
                    if (*nam != *cmp) {
                            break;
```

```
                }
                nam++;
                cmp++;
        }
        if ( (*nam==NULLCH) && (*cmp==NULLCH) ) { /* Name found */
                found = TRUE;
                break;
        }
}

/* Case #1 - file is not in directory (i.e., does not exist)    */

if (! found) {
        if (mbits & LF_MODE_O) {        /* File *must* exist    */
                signal(Lf_data.lf_mutex);
                return SYSERR;
        }

        /* Take steps to create new file and add to directory   */

        /* Verify that space remains in the directory */

        if (dirptr->lfd_nfiles >= LF_NUM_DIR_ENT) {
                signal(Lf_data.lf_mutex);
                return SYSERR;
        }

        /* Allocate next dir. entry & initialize to empty file  */

        ldptr = &dirptr->lfd_files[dirptr->lfd_nfiles++];
        ldptr->ld_size = 0;
        from = name;
        to = ldptr->ld_name;
        while ( (*to++ = *from++) != NULLCH ) {
                ;
        }
        ldptr->ld_ilist = LF_INULL;

/* Case #2 - file is in directory (i.e., already exists)        */

} else if (mbits & LF_MODE_N) {         /* File must not exist  */
                signal(Lf_data.lf_mutex);
                return SYSERR;
}
```

```
        /* Initialize the local file pseudo-device */

        lfptr = &lfltab[lfnext];
        lfptr->lfstate = LF_USED;
        lfptr->lfdirptr = ldptr;        /* Point to directory entry     */
        lfptr->lfmode = mbits & LF_MODE_RW;

        /* File starts at position 0 */

        lfptr->lfpos     = 0;

        to = lfptr->lfname;
        from = name;
        while ( (*to++ = *from++) != NULLCH ) {
                ;
        }

        /* Neither index block nor data block are initially valid       */

        lfptr->lfinum    = LF_INULL;
        lfptr->lfdnum    = LF_DNULL;

        /* Initialize byte pointer to address beyond the end of the     */
        /*      buffer (i.e., invalid pointer triggers setup)           */

        lfptr->lfbyte = &lfptr->lfdblock[LF_BLKSIZ];
        lfptr->lfibdirty = FALSE;
        lfptr->lfdbdirty = FALSE;

        signal(Lf_data.lf_mutex);

        return lfptr->lfdev;
}
```

After verifying that the length of the file name is valid, *lfsopen* calls *lfgetmode* to parse the *mode* argument and convert it to a set of bits. A mode argument consists of a null-terminated string that contains zero or more of the characters from Figure 19.5. Characters in the mode string may not be repeated, and the combination of "o" and "n" (i.e., file is both *old* and *new*) is considered illegal. Furthermore, if neither "r" nor "w" is present in the mode string, *lfgetmode* assumes a default mode that allows both reading and writing. After parsing the mode string, *lfgetmode* produces an integer with individual bits specifying the modes as defined in *lfilesys.h*.† File *lfgetmode.c* contains the code.

†File *lfilesys.h* can be found on page 464.

Character	Meaning
r	Open the file for reading
w	Open the file for writing
o	File must be “old” (i.e., must exist)
n	File must be “new” (i.e., must not exist)

Figure 19.5 Characters permitted in a mode string and their meaning.

```
/* lfgetmode.c - lfgetmode */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lfgetmode  -  Parse mode argument and generate integer of mode bits
 *------------------------------------------------------------------------
 */
int32   lfgetmode (
         char   *mode                   /* String of mode characters    */
        )
{
        int32   mbits;                  /* Mode bits to return          */
        char    ch;                     /* Next char in mode string     */

        mbits = 0;

        /* Mode string specifies:                                       */
        /*      r - read                                                */
        /*      w - write                                               */
        /*      o - old (file must exist)                               */
        /*      n - new (create a new file)                             */

        while ( (ch = *mode++) != NULLCH) {
                switch (ch) {

                    case 'r':   if (mbits&LF_MODE_R) {
                                        return SYSERR;
                                }
                                mbits |= LF_MODE_R;
                                continue;
```

```
                case 'w':   if (mbits&LF_MODE_W) {
                                    return SYSERR;
                            }
                            mbits |= LF_MODE_W;
                            continue;

                case 'o':   if (mbits&LF_MODE_O || mbits&LF_MODE_N) {
                                    return SYSERR;
                            }
                            mbits |= LF_MODE_O;
                            break;

                case 'n':   if (mbits&LF_MODE_O || mbits&LF_MODE_N) {
                                    return SYSERR;
                            }
                            mbits |= LF_MODE_N;
                            break;

                default:    return SYSERR;
            }
        }

        /* If neither read nor write specified, allow both */

        if ( (mbits&LF_MODE_RW) == 0 ) {
                mbits |= LF_MODE_RW;
        }
        return mbits;
}
```

Once the mode argument has been parsed, *lfsopen* verifies that the file is not already open, verifies that a file pseudo-device is available, and searches the file system directory to see if the file exists. If the file exists (and the mode allows opening an existing file), *lfsopen* fills in the control block of a file pseudo-device. If the file does not exist (and the mode allows creating a new file), *lfsopen* allocates an entry in the file system directory, and then fills in the control block of a file pseudo-device. The initial file position is set to zero. Control block fields *lfinum* and *lfdnum* are set to the appropriate null value to indicate that neither the index block nor the data block are currently present. More important, field *lfbyte* is set to a value beyond the end of the data block buffer. We will see that setting *lfbyte* is important because the code uses the following invariant when accessing data:

When lfbyte *contains an address in* lfdblock*, the byte to which it points contains the data that is in the file at the position given by* lfpos*; when* lfbyte *contains an address beyond* lfdblock*, values in* lfdblock *cannot be used.*

The general idea is that setting the current byte position beyond the end of the current data block causes the system to fetch another data block before performing transfers. That is, any attempt to read or write bytes will cause the system to read a new data block from disk. The details will become clear when we examine data transfer functions, such as *lflgetc* and *lflputc*.

19.14 Closing A File Pseudo-Device (lflclose)

When an application finishes using a file, the application calls *close* to terminate use. Note that the application has been using one of the pseudo-devices to read and write the file. Therefore, when the application calls *close*, the device descriptor corresponds to a pseudo-device. Closing the device terminates use of the file, and makes the pseudo-device available for other uses.

In theory, closing a pseudo-device is trivial: change the state to indicate that the device is no longer in use. In practice, however, caching complicates closing because the control block may contain data that has not been written to the file. Therefore, function *lflclose* must check the *dirty bits*† in the control block that specify whether the index block or data block have been changed since they were written to disk.

If changes have occurred since the last time information was written to disk, the data block (and possibly the index block) should be written to disk before the control block is deallocated and made available for another file. Rather than write data directly, *lflclose* calls function *lfflush* to take care of writing the items to disk. Isolating all updates to *lfflush* keeps the design cleaner. Once the disk has been updated, *lflclose* changes the state of the control block. File *lflclose.c* contains the code.

19.15 Flushing Data To Disk (lfflush)

Function *lfflush* operates as expected. It receives a pointer to the control block of the pseudo-device as an argument, and uses the pointer to examine "dirty" bits in the control block. If the index block has changed, *lfflush* uses *lfibput* to write a copy to disk; if the data block has changed, *lfflush* uses *write* to write a copy to disk. Fields *lfinum* and *lfdnum* contain the index block number and data block number to use. The code can be found in file *lfflush.c*.

†The term *dirty bit* refers to a Boolean (i.e., a single bit) that is set to *TRUE* to indicate that data has changed.

```
/* lflclose.c - lflclose.c */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lflclose  -  Close a file by flushing output and freeing device entry
 *------------------------------------------------------------------------
 */
devcall lflclose (
          struct dentry *devptr         /* Entry in device switch table */
        )
{
        struct  lflcblk *lfptr;         /* Ptr to open file table entry */

        /* Obtain exclusive use of the file */

        lfptr = &lfltab[devptr->dvminor];
        wait(lfptr->lfmutex);

        /* If file is not open, return an error */

        if (lfptr->lfstate != LF_USED) {
                signal(lfptr->lfmutex);
                return SYSERR;
        }

        /* Write index or data blocks to disk if they have changed */

        if (Lf_data.lf_dirdirty || lfptr->lfdbdirty || lfptr->lfibdirty) {
                lfflush(lfptr);
        }

        /* Set device state to FREE and return to caller */

        lfptr->lfstate = LF_FREE;
        signal(lfptr->lfmutex);
        return OK;
}
```

```
/* lfflush.c - lfflush */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lfflush  -  Flush directory, data block, and index block for an open
 *                      file (assumes file mutex is held)
 *------------------------------------------------------------------------
 */
status  lfflush (
          struct lflcblk  *lfptr        /* Ptr to file pseudo device    */
        )
{

        if (lfptr->lfstate == LF_FREE) {
                return SYSERR;
        }

        /* Write the directory if it has changed */

        if (Lf_data.lf_dirdirty) {
                write(Lf_data.lf_dskdev, (char *)&Lf_data.lf_dir,
                                                        LF_AREA_DIR);
                Lf_data.lf_dirdirty = FALSE;
        }

        /* Write data block if it has changed */

        if (lfptr->lfdbdirty) {
                write(Lf_data.lf_dskdev, lfptr->lfdblock, lfptr->lfdnum);
                lfptr->lfdbdirty = FALSE;
        }

        /* Write i-block if it has changed */

        if (lfptr->lfibdirty) {
                lfibput(Lf_data.lf_dskdev, lfptr->lfinum, &lfptr->lfiblock);
                lfptr->lfibdirty = FALSE;
        }

        return OK;
}
```

19.16 Bulk Transfer Functions For A File (lflwrite, lflread)

Our implementation adopts a straightforward approach to transfer operations *write* and *read*: each uses a loop that calls a character transfer function. For example, function *lflwrite* calls *lflputc* repeatedly. File *lflwrite.c* contains the code.

```
/* lflwrite.c - lflwrite */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lflwrite  --  Write data to a previously opened local disk file
 *------------------------------------------------------------------------
 */
devcall lflwrite (
          struct dentry *devptr,        /* Entry in device switch table */
          char  *buff,                  /* Buffer holding data to write */
          int32 count                   /* Number of bytes to write     */
        )
{
        int32           i;              /* Number of bytes written      */

        if (count < 0) {
                return SYSERR;
        }

        /* Iteratate and write one byte at a time */

        for (i=0; i<count; i++) {
                if (lflputc(devptr, *buff++) == SYSERR) {
                        return SYSERR;
                }
        }
        return count;
}
```

Function *lflread* calls *lflgetc* repeatedly to satisfy a *read* request. It places each byte in the next location of the caller's buffer. An interesting part of the code concerns how *lflread* handles an *end-of-file* condition. When it reaches the end of the file, *lflgetc* returns constant *EOF*. If *lflread* has already extracted one or more bytes of data from the file when it receives an *EOF* from *lflgetc*, *lflread* stops the loop and returns a count of the bytes that have been read. If no data has been found when an end-of-file is received, *lflread* returns constant *EOF* to its caller. File *lflread.c* contains the code.

```
/* lflread.c - lflread */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lflread  -  Read from a previously opened local file
 *------------------------------------------------------------------------
 */
devcall lflread (
          struct dentry *devptr,        /* Entry in device switch table */
          char  *buff,                  /* Buffer to hold bytes         */
          int32 count                   /* Max bytes to read            */
        )
{
        uint32  numread;                /* Number of bytes read         */
        int32   nxtbyte;                /* Character or SYSERR/EOF      */

        if (count < 0) {
                return SYSERR;
        }

        /* Iterate and use lflgetc to read indivdiual bytes */

        for (numread=0 ; numread < count ; numread++) {
                nxtbyte = lflgetc(devptr);
                if (nxtbyte == SYSERR) {
                        return SYSERR;
                } else if (nxtbyte == EOF) {    /* EOF before finished */
                    if (numread == 0) {
                        return EOF;
                    } else {
                        return numread;
                    }
                } else {
                        *buff++ = (char) (0xff & nxtbyte);
                }
        }
        return numread;
}
```

19.17 Seeking To A New Position In the File (lflseek)

A process can call *seek* to change the current position in a file. Our system uses function *lflseek* to implement *seek*, and restricts the position to a valid point in the file (i.e., it is unlike Unix, which allows an application to seek beyond the end of the file).

Seeking to a new position consists of changing field *lfpos* in the file control block and setting field *lfbyte* to an address beyond *lfdblock* (which, according to the invariant above, means that the pointer cannot be used to extract data until the index block and data block are in place). File *lflseek.c* contains the code.

```
/* lflseek.c - lflseek */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lflseek  -  Seek to a specified position in a file
 *------------------------------------------------------------------------
 */
devcall lflseek (
          struct dentry *devptr,        /* Entry in device switch table */
          uint32        offset          /* Byte position in the file    */
        )
{
        struct  lflcblk *lfptr;         /* Ptr to open file table entry */

        /* If file is not open, return an error */

        lfptr = &lfltab[devptr->dvminor];
        wait(lfptr->lfmutex);
        if (lfptr->lfstate != LF_USED) {
                signal(lfptr->lfmutex);
                return SYSERR;
        }

        /* Verify offset is within current file size */

        if (offset > lfptr->lfdirptr->ld_size) {
                signal(lfptr->lfmutex);
                return SYSERR;
        }

        /* Record new offset and invalidate byte pointer (i.e., force   */
        /*   the index and data blocks to be replaced if a successive   */
        /*   call is made to read or write)                              */
```

```
        lfptr->lfpos = offset;
        lfptr->lfbyte = &lfptr->lfdblock[LF_BLKSIZ];

        signal(lfptr->lfmutex);
        return OK;
}
```

19.18 Extracting One Byte From A File (lflgetc)

Once a file has been opened and both the correct index block and data block have been loaded into memory, extracting a byte from the file is trivial: it consists of treating *lfbyte* as a pointer to the byte, extracting the byte, and advancing the buffer pointer to the next byte. Function *lflgetc*, which performs the operation, can be found in file *lflgetc.c*.

```
/* lflgetc.c - lfgetc */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lflgetc  -  Read the next byte from an open local file
 *------------------------------------------------------------------------
 */
devcall lflgetc (
          struct dentry *devptr         /* Entry in device switch table */
        )
{
        struct  lflcblk *lfptr;         /* Ptr to open file table entry */
        struct  ldentry *ldptr;         /* Ptr to file's entry in the   */
                                        /*   in-memory directory        */
        int32   onebyte;                /* Next data byte in the file   */

        /* Obtain exclusive use of the file */

        lfptr = &lfltab[devptr->dvminor];
        wait(lfptr->lfmutex);

        /* If file is not open, return an error */

        if (lfptr->lfstate != LF_USED) {
                signal(lfptr->lfmutex);
                return SYSERR;
        }
```

```
        /* Return EOF for any attempt to read beyond the end-of-file */

        ldptr = lfptr->lfdirptr;
        if (lfptr->lfpos >= ldptr->ld_size) {
                signal(lfptr->lfmutex);
                return EOF;
        }

        /* If byte pointer is beyond the current data block, set up     */
        /*      a new data block                                         */

        if (lfptr->lfbyte >= &lfptr->lfdblock[LF_BLKSIZ]) {
                lfsetup(lfptr);
        }

        /* Extract the next byte from block, update file position, and  */
        /*      return the byte to the caller                           */

        onebyte = 0xff & *lfptr->lfbyte++;
        lfptr->lfpos++;
        signal(lfptr->lfmutex);
        return onebyte;
}
```

If the specified file is not open, *lflgetc* returns *SYSERR*. If the current file position exceeds the file size, *lflgetc* returns *EOF*. In other cases, *lflgetc* checks pointer *lfbyte* to see whether it currently points outside of the data block in *lfdblock*. If so, *lflgetc* calls *lfsetup* to read the correct index block and data block into memory.

Once the data block is in memory, *lflgetc* can extract a byte. To do so, it dereferences *lfbyte* to obtain a byte and places the byte in variable *onebyte*. It increments both the byte pointer and the file position before returning the byte to the caller.

19.19 Changing One Byte In A File (lflputc)

Function *lflputc* handles the task of storing a byte into a file at the current position. As with *lflgetc*, the code that performs the transfer is trivial and only occupies a few lines. Pointer *lfbyte* gives a location in *lfdblock* at which the byte should be stored; the code uses the pointer, stores the specified byte, increments the pointer, and sets *lfdbdirty* to indicate that the data block has been changed. Note that *lflputc* merely accumulates characters in a buffer in memory; it does not write the buffer to disk each time a change occurs. The buffer will only be copied to disk when the current position moves to another disk block.

```
/* lflputc.c - lfputc */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lflputc  -  Write a single byte to an open local file
 *------------------------------------------------------------------------
 */
devcall lflputc (
          struct dentry *devptr,        /* Entry in device switch table */
          char          ch              /* Character (byte) to write    */
        )
{
        struct  lflcblk *lfptr;         /* Ptr to open file table entry */
        struct  ldentry *ldptr;         /* Ptr to file's entry in the   */
                                        /*  in-memory directory         */

        /* Obtain exclusive use of the file */

        lfptr = &lfltab[devptr->dvminor];
        wait(lfptr->lfmutex);

        /* If file is not open, return an error */

        if (lfptr->lfstate != LF_USED) {
                signal(lfptr->lfmutex);
                return SYSERR;
        }

        /* Return SYSERR for an attempt to skip bytes beyond the byte   */
        /*      that is currently the end of the file                   */

        ldptr = lfptr->lfdirptr;
        if (lfptr->lfpos > ldptr->ld_size) {
                signal(lfptr->lfmutex);
                return SYSERR;
        }

        /* If pointer is outside current block, set up new block */

        if (lfptr->lfbyte >= &lfptr->lfdblock[LF_BLKSIZ]) {

                /* Set up block for current file position */

                lfsetup(lfptr);
        }
```

```
        /* If appending a byte to the file, increment the file size.    */
        /*   Note: comparison might be equal, but should not be greater.*/

        if (lfptr->lfpos >= ldptr->ld_size) {
                ldptr->ld_size++;
                Lf_data.lf_dirdirty = TRUE;
        }

        /* Place byte in buffer and mark buffer "dirty" */

        *lfptr->lfbyte++ = ch;
        lfptr->lfpos++;
        lfptr->lfdbdirty = TRUE;

        signal(lfptr->lfmutex);
        return OK;
}
```

Like *lflgetc*, *lflputc* examines *lfbyte* on each call. If *lfbyte* lies outside of data block *lfdblock*, *lflputc* calls *lfsetup* to move to the next block. However, a subtle difference exists in the way *lflputc* and *lflgetc* treat an invalid file position. *Lflgetc* always returns *EOF* if the file position exceeds the last byte of the file. *Lflputc* returns *SYSERR* when the file position is more than one byte beyond the end of the file, but if the position is exactly one byte beyond the end, it allows the operation to proceed. That is, it allows a file to be extended. When a file is extended, the file size, found in the directory entry for the file, must be incremented. File *lflputc.c* contains the code.

19.20 Loading An Index Block And A Data Block (lfsetup)

Once a file position has been assigned to field *lfpos*, function *lfsetup* handles the details of positioning the in-memory data structures at the specified position. That is, *lfsetup* identifies the index block for the position, loads a copy of the index block into memory, identifies the data block that contains the position, and loads a copy of the data block into memory. *Lfsetup* begins by obtaining pointers to data structures.

A key idea is that *lfsetup* handles the case where an existing data block is dirty. To do so, before it reads new items from disk, *lfsetup* checks the control block. If the existing index or data blocks have changed, *lfsetup* calls *lfflush* to write them to disk.

The first step in loading data for the current file position consists of finding the index block that covers the specified file position. To do so, *lfsetup* starts with an index block that precedes the current position, and moves along the linked list of index blocks to find the needed block. There are two cases for an initial block: no index block has been loaded or an index block is currently in memory.

If no index block has been loaded (i.e., the file was just opened), *lfsetup* obtains one. For a new file, *lfsetup* must allocate an initial index block from the free list; for an existing file, it loads the first index block for the file. In either case, the initial condition will be that the index block either corresponds to the file position or covers an earlier position in the file.

If an index block has already been loaded, there are two possibilities. If the index block covers an earlier part of the file, no further action is needed. If the index block corresponds to a portion of the file that lies after the current file position (e.g., a process has issued a *seek* to an earlier position), *lfsetup* must find an initial block that covers an earlier part of the file. To handle the case, *lfsetup* replaces the index block with the initial index block for the file.

Once it has loaded an index block that covers an earlier part of the file, *lfsetup* enters a loop that moves along the linked list of index blocks until it reaches the index block that covers the current file position. At each iteration, *lfsetup* uses field *ib_next* to find the number of the next index block on the list, and then calls *lfibget* to read the index block into memory.

Once the correct index block has been loaded, *lfsetup* must determine the data block to load. To do so, it uses the file position to compute an index (from 0 through 15) for the data block array. Because each index block covers exactly 8K bytes (i.e., 2^{13} bytes) of data and each slot in the array corresponds to a block of 512 bytes (2^9), binary arithmetic can be used. *Lfsetup* computes the *logical and* of the *LF_IMASK* (the low-order 13 bits), and then shifts the result right 9 bits.

Lfsetup uses the result of the above computation as an index into array *ib_dba* to obtain the ID of a data block. There are two cases that require *lfsetup* to load a new data block. In the first case, the pointer in the array is null, which means *lflputc* is about to write a new byte on the end of the file and no data block has been assigned for the position. *Lfsetup* calls *lfdballoc* to allocate a new data block from the free list, and records the ID in the entry of array *ib_dba*. In the second case, the entry in array *ib_dba* specifies a data block other than the data block currently loaded. *Lfsetup* calls *read* to fetch the correct data block from disk.

As the final step before returning, *lfsetup* uses the file position to compute a position within the data block, and assigns the address to field *lfbyte*. The careful arrangement of making the data block size a power of two means that the indices from 0 through 511 can be computed by selecting the low-order 9 bits of the file position. The code uses a *logical and* with mask *LF_DMASK*. File *lfsetup.c* contains the code.

```
/* lfsetup.c - lfsetup */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lfsetup  -  Set a file's index block and data block for the current
 *               file position (assumes file mutex held)
 *------------------------------------------------------------------------
 */
status  lfsetup (
          struct lflcblk  *lfptr        /* Pointer to slave file device */
        )
{
        dbid32  dnum;                   /* Data block to fetch          */
        ibid32  ibnum;                  /* I-block number during search */
        struct  ldentry *ldptr;         /* Ptr to file entry in dir.    */
        struct  lfiblk  *ibptr;         /* Ptr to in-memory index block */
        uint32  newoffset;              /* Computed data offset for     */
                                        /*   next index block           */
        int32   dindex;                 /* Index into array in an index */
                                        /*   block                      */


        /* Obtain exclusive access to the directory */

        wait(Lf_data.lf_mutex);

        /* Get pointers to in-memory directory, file's entry in the     */
        /*      directory, and the in-memory index block                */

        ldptr = lfptr->lfdirptr;
        ibptr = &lfptr->lfiblock;

        /* If existing index block or data block changed, write to disk */

        if (lfptr->lfibdirty || lfptr->lfdbdirty) {
                lfflush(lfptr);
        }
        ibnum = lfptr->lfinum;          /* Get ID of curr. index block  */

        /* If there is no index block in memory (e.g., because the file */
        /*      was just opened), either load the first index block of  */
        /*      the file or allocate a new first index block            */

        if (ibnum == LF_INULL) {
```

```
        /* Check directory entry to see if index block exists   */

        ibnum = ldptr->ld_ilist;
        if (ibnum == LF_INULL) { /* Empty file - get new i-block*/
                ibnum = lfiballoc();
                lfibclear(ibptr, 0);
                ldptr->ld_ilist = ibnum;
                lfptr->lfibdirty = TRUE;
        } else {                /* Nonempty - read first i-block*/
                lfibget(Lf_data.lf_dskdev, ibnum, ibptr);
        }
        lfptr->lfinum = ibnum;

/* Otherwise, if current file position has been moved to an     */
/*   offset before the current index block, start at the        */
/*   beginning of the index list for the file                    */

} else if (lfptr->lfpos < ibptr->ib_offset) {

        /* Load initial index block for the file (we know that  */
        /*      at least one index block exists)                */

        ibnum = ldptr->ld_ilist;
        lfibget(Lf_data.lf_dskdev, ibnum, ibptr);
        lfptr->lfinum = ibnum;
}

/* At this point, an index block is in memory, but may cover    */
/*  an offset less than the current file position.  Loop until  */
/*  the index block covers the current file position.           */

while ((lfptr->lfpos & ~LF_IMASK) > ibptr->ib_offset ) {
        ibnum = ibptr->ib_next;
        if (ibnum == LF_INULL) {
                /* Allocate new index block to extend file */
                ibnum = lfiballoc();
                ibptr->ib_next = ibnum;
                lfibput(Lf_data.lf_dskdev, lfptr->lfinum, ibptr);
                lfptr->lfinum = ibnum;
                newoffset = ibptr->ib_offset + LF_IDATA;
                lfibclear(ibptr, newoffset);
                lfptr->lfibdirty = TRUE;
        } else {
                lfibget(Lf_data.lf_dskdev, ibnum, ibptr);
                lfptr->lfinum = ibnum;
```

```
                }
                lfptr->lfdnum = LF_DNULL; /* Invalidate old data block */
        }

        /* At this point, the index block in lfiblock covers the        */
        /*  current file position (i.e., position lfptr->lfpos).  The   */
        /*  next step consists of loading the correct data block.       */

        dindex = (lfptr->lfpos & LF_IMASK) >> 9;

        /* If data block index does not match current data block, read  */
        /*   the correct data block from disk                           */

        dnum = lfptr->lfiblock.ib_dba[dindex];
        if (dnum == LF_DNULL) {         /* Allocate new data block */
                dnum = lfdballoc((struct lfdbfree *)&lfptr->lfdblock);
                lfptr->lfiblock.ib_dba[dindex] = dnum;
                lfptr->lfibdirty = TRUE;
        } else if ( dnum != lfptr->lfdnum) {
                read(Lf_data.lf_dskdev, (char *)lfptr->lfdblock, dnum);
                lfptr->lfdbdirty = FALSE;
        }
        lfptr->lfdnum = dnum;

        /* Use current file offset to set the pointer to the next byte  */
        /*   within the data block                                      */

        lfptr->lfbyte = &lfptr->lfdblock[lfptr->lfpos & LF_DMASK];
        signal(Lf_data.lf_mutex);
        return OK;
}
```

19.21 Master File System Device Initialization (lfsinit)

Initialization for the master file system device is straightforward. Function *lfsinit* performs the task, which consists of recording the ID of the disk device, creating a semaphore that provides mutual exclusion to the directory, clearing the in-memory directory (merely to help with debugging), and setting a Boolean to indicate that the directory has not been read into memory. Data for the master file system device is kept in the global structure *Lf_data*. File *lfsinit.c* contains the code.

```
/* lfsinit.c - lfsinit */

#include <xinu.h>

struct  lfdata  Lf_data;

/*------------------------------------------------------------------------
 * lfsinit  -  Initialize the local file system master device
 *------------------------------------------------------------------------
 */
devcall lfsinit (
          struct dentry *devptr         /* Entry in device switch table */
        )
{
        /* Assign ID of disk device that will be used */

        Lf_data.lf_dskdev = LF_DISK_DEV;

        /* Create a mutual exclusion semaphore */

        Lf_data.lf_mutex = semcreate(1);

        /* Zero directory area (for debugging) */

        memset((char *)&Lf_data.lf_dir, NULLCH, sizeof(struct lfdir));

        /* Initialize directory to "not present" in memory */

        Lf_data.lf_dirpresent = Lf_data.lf_dirdirty = FALSE;

        return OK;
}
```

19.22 Pseudo-Device Initialization (lflinit)

When it opens a file, *lfsopen* initializes many of the entries in the control block for the file pseudo-device. However, some initialization is performed at system startup. To indicate that the device is not in use, the state is assigned *LF_FREE*. A mutual exclusion semaphore is created to guarantee that at most one operation will be in progress on the file at a given time. Most other fields in the control block are initialized to zero (they will not be used until a file is opened, but initializing to zero can make debugging easier). File *lflinit.c* contains the code.

```
/* lflinit.c - lflinit */

#include <xinu.h>

struct  lflcblk lfltab[Nlfl];           /* Pseudo-device control blocks */

/*------------------------------------------------------------------------
 * lflinit  -  Initialize control blocks for local file pseudo-devices
 *------------------------------------------------------------------------
 */
devcall lflinit (
          struct dentry *devptr         /* Entry in device switch table */
        )
{
        struct  lflcblk *lfptr;         /* Ptr. to control block entry  */
        int32   i;                      /* Walks through name array     */

        lfptr = &lfltab[ devptr->dvminor ];

        /* Initialize control block entry */

        lfptr->lfstate = LF_FREE;       /* Device is currently unused   */
        lfptr->lfdev = devptr->dvnum;   /* Set device ID                */
        lfptr->lfmutex = semcreate(1);  /* Create the mutex semaphore   */

        /* Initialize the directory and file position */

        lfptr->lfdirptr = (struct  ldentry *) NULL;
        lfptr->lfpos = 0;
        for (i=0; i<LF_NAME_LEN; i++) {
                lfptr->lfname[i] = NULLCH;
        }

        /* Zero the in-memory index block and data block */

        lfptr->lfinum = LF_INULL;
        memset((char *) &lfptr->lfiblock, NULLCH, sizeof(struct lfiblk));
        lfptr->lfdnum = 0;
        memset((char *) &lfptr->lfdblock, NULLCH, LF_BLKSIZ);

        /* Start with the byte beyond the current data block */

        lfptr->lfbyte = &lfptr->lfdblock[LF_BLKSIZ];
        lfptr->lfibdirty = lfptr->lfdbdirty = FALSE;
        return OK;
}
```

19.23 File Truncation (lftruncate)

We will use file truncation as a way to show how file data structures are deallocated. To truncate a file to zero length, each of the index blocks for the file must be placed on the free list of index blocks. Before an index block can be released, however, each of the data blocks to which the index block points must be placed on the free list of data blocks. Function *lftruncate* performs file truncation; file *lftruncate.c* contains the code.

```
/* lftruncate.c - lftruncate */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lftruncate  -  Truncate a file by freeing its index and data blocks
 *                      (assumes directory mutex held)
 *------------------------------------------------------------------------
 */
status  lftruncate (
          struct lflcblk *lfptr         /* Ptr to file's cntl blk entry */
        )
{
        struct  ldentry *ldptr;         /* Pointer to file's dir. entry */
        struct  lfiblk  iblock;         /* Buffer for one index block   */
        ibid32  ifree;                  /* Start of index blk free list */
        ibid32  firstib;                /* First index blk of the file  */
        ibid32  nextib;                 /* Walks down list of the       */
                                        /*   file's index blocks        */
        dbid32  nextdb;                 /* Next data block to free      */
        int32   i;                      /* Moves through data blocks in */
                                        /*   a given index block        */

        ldptr = lfptr->lfdirptr;        /* Get pointer to dir. entry    */
        if (ldptr->ld_size == 0) {      /* File is already empty */
                return OK;
        }

        /* Clean up the open local file first */

        if ( (lfptr->lfibdirty) || (lfptr->lfdbdirty) ) {
                lfflush(lfptr);
        }
        lfptr->lfpos = 0;
        lfptr->lfinum = LF_INULL;
```

```
lfptr->lfdnum = LF_DNULL;
lfptr->lfbyte = &lfptr->lfdblock[LF_BLKSIZ];

/* Obtain ID of first index block on free list */

ifree = Lf_data.lf_dir.lfd_ifree;

/* Record file's first i-block and clear directory entry */

firstib = ldptr->ld_ilist;
ldptr->ld_ilist = LF_INULL;
ldptr->ld_size = 0;
Lf_data.lf_dirdirty = TRUE;

/* Walk along index block list, disposing of each data block    */
/*   and clearing the corresponding pointer.  A note on loop    */
/*   termination: last pointer is set to ifree below.           */

for (nextib=firstib; nextib!=ifree; nextib=iblock.ib_next) {

        /* Obtain a copy of current index block from disk       */

        lfibget(Lf_data.lf_dskdev, nextib, &iblock);

        /* Free each data block in the index block              */

        for (i=0; i<LF_IBLEN; i++) {    /* For each d-block     */

                /* Free the data block */

                nextdb = iblock.ib_dba[i];
                if (nextdb != LF_DNULL) {
                        lfdbfree(Lf_data.lf_dskdev, nextdb);
                }

                /* Clear entry in i-block for this d-block      */

                        iblock.ib_dba[i] = LF_DNULL;
        }

        /* Clear offset (just to make debugging easier)         */

        iblock.ib_offset = 0;

        /* For the last index block on the list, make it point  */
```

```
                /*      to the current free list                        */

                if (iblock.ib_next == LF_INULL) {
                        iblock.ib_next = ifree;
                }

                /* Write cleared i-block back to disk */

                lfibput(Lf_data.lf_dskdev, nextib, &iblock);
        }

        /* Last index block on the file list now points to first node   */
        /*   on the current free list.  Once we make the free list      */
        /*   point to the first index block on the file list, the       */
        /*   entire set of index blocks will be on the free list        */

        Lf_data.lf_dir.lfd_ifree = firstib;

        /* Indicate that directory has changed and return */

        Lf_data.lf_dirdirty = TRUE;

        return OK;
}
```

The approach used is straightforward: if the file already has length zero, return to the caller. Otherwise, walk along the file's index block list. Read each index block into memory, and call *lfdbfree* to free each data block to which the index block points.

Once the final index block has been reached, add all the index blocks for the file to the free list. To do so, observe that all index blocks for the file are already linked. Thus, only two pointer changes are needed. First, change the *next* pointer in the final index block of the file to point to the current free list. Second, change the free list to point to the first index block of the file.

19.24 Initial File System Creation (lfscreate)

A final initialization function will complete details of the file system. Function *lfscreate* creates an initial, empty file system on a disk. That is, it forms a free list of index blocks, a free list of data blocks, and a directory that contains no files. File *lfscreate.c* contains the code.

```
/* lfscreate.c - lfscreate */

#include <xinu.h>
#include <ramdisk.h>

/*------------------------------------------------------------------------
 * lfscreate  -  Create an initially-empty file system on a disk
 *------------------------------------------------------------------------
 */
status  lfscreate (
          did32         disk,           /* ID of an open disk device    */
          ibid32        lfiblks,        /* Num. of index blocks on disk */
          uint32        dsiz            /* Total size of disk in bytes  */
        )
{
        uint32  sectors;                /* Number of sectors to use     */
        uint32  ibsectors;              /* Number of sectors of i-blocks*/
        uint32  ibpersector;            /* Number of i-blocks per sector*/
        struct  lfdir   dir;            /* Buffer to hold the directory */
        uint32  dblks;                  /* Total free data blocks       */
        struct  lfiblk  iblock;         /* Space for one i-block        */
        struct  lfdbfree dblock;        /* Data block on the free list  */
        dbid32  dbindex;                /* Index for data blocks        */
        int32   retval;                 /* Return value from func call  */
        int32   i;                      /* Loop index                   */

        /* Compute total sectors on disk */

        sectors = dsiz  / LF_BLKSIZ;    /* Truncate to full sector */

        /* Compute number of sectors comprising i-blocks */

        ibpersector = LF_BLKSIZ / sizeof(struct lfiblk);
        ibsectors = (lfiblks+(ibpersector-1)) / ibpersector;/* Round up */
        lfiblks = ibsectors * ibpersector;
        if (ibsectors > sectors/2) {    /* Invalid arguments */
                return SYSERR;
        }

        /* Create an initial directory */

        memset((char *)&dir, NULLCH, sizeof(struct lfdir));
        dir.lfd_nfiles = 0;
        dbindex= (dbid32)(ibsectors + 1);
        dir.lfd_dfree = dbindex;
```

```
        dblks = sectors - ibsectors - 1;
        retval = write(disk,(char *)&dir, LF_AREA_DIR);
        if (retval == SYSERR) {
                return SYSERR;
        }

        /* Create list of free i-blocks on disk */

        lfibclear(&iblock, 0);
        for (i=0; i<lfiblks-1; i++) {
                iblock.ib_next = (ibid32)(i + 1);
                lfibput(disk, i, &iblock);
        }
        iblock.ib_next = LF_INULL;
        lfibput(disk, i, &iblock);

        /* Create list of free data blocks on disk */

        memset((char*)&dblock, NULLCH, LF_BLKSIZ);
        for (i=0; i<dblks-1; i++) {
                dblock.lf_nextdb = dbindex + 1;
                write(disk, (char *)&dblock, dbindex);
                dbindex++;
        }
        dblock.lf_nextdb = LF_DNULL;
        write(disk, (char *)&dblock, dbindex);
        close(disk);
        return OK;
}
```

19.25 Perspective

File systems are among the most complex pieces of an operating system. Our implementation avoids one of the most challenging problems, sharing, by imposing a restriction: a file can only be opened once at a given time. If the restriction is relaxed, a file system must coordinate operations among multiple file descriptors that can refer to the same file. Sharing raises the question of semantics: how should overlapping *write* operations be interpreted? In particular, if a process attempts to write bytes 0 through N of a file and another process simultaneously attempts to write bytes 2 through N-1 of the same file, what should happen? Should the file system guarantee that one of the two operations occurs first? Should the file system allow bytes to be intermingled? How should the file system manage a shared cache to make operations efficient?

A second form of complexity arises from the implementation. All operations on files must be translated into operations on disk blocks. As a result, basic data structures, such as linked lists, can be complex to manipulate. Interestingly, much of the complexity arises when disk blocks are shared. For example, because a given disk block can hold index blocks from multiple files, two processes may need to access the same disk block simultaneously. Most file systems arrange to cache disk blocks, making such access efficient.

A final form of complexity arises from the need for safety and recovery. Users assume that once data has been written to a file, the data is "safe," even if the power fails. However, a file system cannot afford to write to disk each time an application stores a byte in a file. Thus, one of the grand challenges of file system design arises from the tradeoff between efficiency and safety — a designer looks for ways to minimize the risk of losing data while also looking for ways to maximize efficiency.

19.26 Summary

A file system manages objects on nonvolatile storage. To ensure the interface to files is the same as the interface to devices, our example system is organized into a master file system device and a set of file pseudo-devices. To access a file, a process *opens* the master device; the call returns the descriptor of a pseudo-device for the file. Once a file has been opened, functions *read*, *write*, *getc*, *putc*, *seek*, and *close* can be used on the file.

Our design allows files to grow dynamically; the data structures for a file consist of a directory entry and a linked list of index blocks that each point to a set of data blocks. When a file is used, the driver software loads an index block and data block into memory. Subsequent accesses or changes to a file affect the data block in memory. When the file position moves outside the current block, the file system writes the data block back to disk and allocates another data block. Similarly, when the file position moves outside the data covered by the current index block, the system writes the current index block to disk and allocates a new index block.

EXERCISES

19.1 Consider the amount of code needed for the Xinu file system. Can you find an alternative file system design that provides equivalent functionality, but requires substantially less code?

19.2 Redesign functions *lflread* and *lflwrite* to perform high-speed copies (i.e., copy bytes from or to the current data block without making repeated calls to *lflgetc* or *lflputc*.

19.3 Redesign the system to permit multiple processes to open the same file simultaneously. Coordinate all writes to ensure that a given byte in the file always contains the data written last.

19.4 Free data blocks are chained together on a singly-linked list. Redesign the system to place them in a file (i.e., reserve index block 0 to be an unnamed "file" in which index blocks point to free data blocks). Compare the performance of the new and original designs.

19.5 What are the maximum number of disk accesses necessary to allocate and free a data block under the original design and the new design in the previous exercise?

19.6 The number of index blocks is important because having too many wastes space that could be used for data, while having too few means data blocks will be wasted because there are insufficient index blocks to use them. Given that there are 16 data block pointers in an index block and 7 index blocks fill a disk block, how many index blocks might be needed for a disk of n total blocks if the directory can hold k files?

19.7 Current index block IDs are 32 bits long. Redesign the system to use 16-bit index block IDs. What are the tradeoffs of the two approaches?

19.8 Redesign the system so it closes all files that have been opened by a process when the process terminates.

19.9 Change the system to have a file switch table separate from the device switch table. What are the advantages and disadvantages of each approach?

19.10 After modifying the free list, function *lfiballoc* writes a copy of the directory to disk. As an alternative, *lfiballoc* could mark the directory "dirty" and defer the *write* operation until later. Discuss the advantages and disadvantages of each approach.

19.11 Consider two processes attempting to write to a single file. Suppose one process repeatedly writes 20 bytes of character *A* and the other process repeatedly writes 20 bytes of character *B*. Describe the order in which characters might appear in the file.

19.12 Create a *control* function for the file pseudo-device driver that allows a caller to invoke *lftruncate*.

19.13 Create a *control* function for the master file system device that allows a caller to invoke *lfscreate*.

Chapter Contents

20.1 Introduction, 509
20.2 Remote File Access, 509
20.3 Remote File Semantics, 510
20.4 Remote File Design And Messages, 510
20.5 Remote File Server Communication (rfscomm), 518
20.6 Sending A Basic Message (rfsndmsg), 520
20.7 Network Byte Order, 522
20.8 A Remote File System Using A Device Paradigm, 522
20.9 Opening A Remote File (rfsopen), 524
20.10 Checking The File Mode (rfsgetmode), 527
20.11 Closing A Remote File (rflclose), 528
20.12 Reading From A Remote File (rflread), 529
20.13 Writing To A Remote File (rflwrite), 532
20.14 Seeking On A Remote File (rflseek), 535
20.15 Character I/O On A Remote File (rflgetc, rflputc), 536
20.16 Remote File System Control Functions (rfscontrol), 537
20.17 Initializing The Remote File System (rfsinit, rflinit), 541
20.18 Perspective, 543
20.19 Summary, 543

20

A Remote File Mechanism

Networking makes the far-away the here-and-now.

— Unknown

20.1 Introduction

Chapter 16 discusses a network interface and a device driver that uses the hardware interface to send and receive packets. Chapter 18 considers disk hardware and the block transfer paradigm. Chapter 19 explains how a file system creates high-level abstractions, including dynamic files, and shows how files can be mapped onto a disk.

This chapter expands the discussion of file systems by considering an alternative that uses a remote file server. That is, instead of implementing the file abstraction directly on hardware, the operating system relies on a separate computer called a server. When an application requests a file operation, the operating system sends a request to the server and receives a response. The next chapter extends the discussion by showing how a remote and local file system can be integrated.

20.2 Remote File Access

A remote file access mechanism requires four conceptual pieces. First, an operating system must contain a device driver for a network device (such as an Ethernet). Second, the operating system must also contain protocol software (such as UDP and IP) that handles addressing so the packets can reach the remote server and replies can return. Third, the operating system must have remote file access software that becomes a client (i.e., forms a request, uses the network to send the request to the server and re-

ceive a response, and interprets the response). Whenever a process invokes an I/O operation on a remote file (e.g., *read* or *write*), the remote file access software forms a message that specifies the operation, sends the request to the remote file server, and processes the response. Fourth, a computer on the network must be running a remote file server application that honors each request.

In practice, many questions arise about the design of a remote file access mechanism. What services should a remote file server provide? Should the service permit a client to create hierarchical directories, or should the server only permit a client to create data files? Should the mechanism allow a client to remove files? If two or more clients send requests to a given server, should the files be shared or should each client have its own files? Should a file be cached in the memory of the client machine? For example, when a process reads a byte from a remote file, should the client software request one-thousand bytes and hold the extra bytes in a cache to avoid sending requests to the remote server for successive bytes?

20.3 Remote File Semantics

One of the primary design considerations surrounding remote file systems arises from heterogeneity: the operating systems on the client and server machines may differ. As a result, the file operations available to the remote server may differ from the file operations used on the client machine. For example, because the file server we use runs on a Unix system (e.g., Linux or Solaris), the server supplies functionality from the Unix file system, not from Xinu.

Some of the Xinu file operations map directly to Unix file operations. For example, Xinu uses the same semantics for *read* as Unix — a request specifies a buffer size and *read* specifies the number of data bytes that were placed in the buffer. Similarly, a Xinu *write* operation follows the same semantics as a Unix *write*.

However, Xinu semantics do differ from Unix semantics in many ways. Each Unix file has an owner that is identified by a Unix userid; Xinu does not have userids, and even if it did, they would not align with the userids used by the server. Even small details differ. For example, the *mode* argument used with a Xinu *open* operation allows a caller to specify that the file must be *new* (i.e., must not exist) or that the file must be *old* (i.e., must exist). Unix allows a file to be created, but does not test whether the file already exists. Instead, if the file exists, Unix truncates the file to zero bytes. Thus, to implement the Xinu *new* mode, a remote server running on a Unix system must first test whether the file exists and return an error indication if it does.

20.4 Remote File Design And Messages

Our example remote file system provides basic functionality: a Xinu process can create a file, write data to the file, seek to an arbitrary position in the file, read data from the file, truncate a file, and delete a file. The remote file system also allows a

Xinu process to create or remove directories. However, the system does not permit processes on multiple computers to share files. For each operation, the system defines a request message (sent from a Xinu client to the remote file server) and a response message (sent from the server back to the Xinu client). Each message begins with a common header that specifies the type of the operation, a status value (used in responses to report errors), a sequence number, and the name of a file. Each outgoing request is assigned a unique sequence number, and the remote file software checks a reply to ensure that an incoming reply matches the outgoing request. Our implementation defines a structure for each message type. To avoid nested structure declarations, the code uses a preprocessor definition, *RF_MSG_HDR*, for the header fields, and then includes the header in each struct. File *rfilesys.h* contains the code.

```
/* rfilesys.h - Definitions for remote file system pseudo-devices */

#ifndef Nrfl
#define Nrfl    10
#endif

/* Control block for a remote file pseudo-device */

#define RF_NAMLEN       128             /* Maximum length of file name  */
#define RF_DATALEN      1024            /* Maximum data in read or write*/
#define RF_MODE_R       F_MODE_R        /* Bit to grant read access     */
#define RF_MODE_W       F_MODE_W        /* Bit to grant write access    */
#define RF_MODE_RW      F_MODE_RW       /* Mask for read and write bits */
#define RF_MODE_N       F_MODE_N        /* Bit for "new" mode           */
#define RF_MODE_O       F_MODE_O        /* Bit for "old" mode           */
#define RF_MODE_NO      F_MODE_NO       /* Mask for "n" and "o" bits    */

/* Global data for the remote server */

#ifndef RF_SERVER_IP
#define RF_SERVER_IP    "128.10.3.51"
#endif

#ifndef RF_SERVER_PORT
#define RF_SERVER_PORT  33123
#endif

#ifndef RF_LOC_PORT
#define RF_LOC_PORT     33123
#endif

struct  rfdata  {
        int32   rf_seq;                 /* Next sequence number to use  */
        uint32  rf_ser_ip;              /* Server IP address            */
```

```
        uint16  rf_ser_port;            /* Server UDP port              */
        uint16  rf_loc_port;            /* Local (client) UPD port      */
        int32   rf_udp_slot;            /* UDP slot to use              */
        sid32   rf_mutex;               /* Mutual exclusion for access  */
        bool8   rf_registered;          /* Has UDP port been registered?*/
};

extern  struct  rfdata  Rf_data;

/* Definition of the control block for a remote file pseudo-device      */

#define RF_FREE 0                       /* Entry is currently unused    */
#define RF_USED 1                       /* Entry is currently in use    */

struct  rflcblk {
        int32   rfstate;                /* Entry is free or used        */
        int32   rfdev;                  /* Device number of this dev.   */
        char    rfname[RF_NAMLEN];      /* Name of the file             */
        uint32  rfpos;                  /* Current file position        */
        uint32  rfmode;                 /* Mode: read access, write     */
                                        /*      access or both          */
};

extern  struct  rflcblk rfltab[];       /* Remote file control blocks   */

/* Definitions of parameters used when accessing a remote server        */

#define RF_RETRIES      3               /* Time to retry sending a msg  */
#define RF_TIMEOUT      3000            /* Wait one second for a reply  */

/* Control functions for a remote file pseudo device */

#define RFS_CTL_DEL     F_CTL_DEL       /* Delete a file                */
#define RFS_CTL_TRUNC   F_CTL_TRUNC     /* Truncate a file              */
#define RFS_CTL_MKDIR   F_CTL_MKDIR     /* Make a directory             */
#define RFS_CTL_RMDIR   F_CTL_RMDIR     /* Remove a directory           */
#define RFS_CTL_SIZE    F_CTL_SIZE      /* Obtain the size of a file    */

/************************************************************************/
/*                                                                      */
/*      Definition of messages exchanged with the remote server         */
/*                                                                      */
/************************************************************************/

/* Values for the type field in messages */
```

```
#define RF_MSG_RESPONSE 0x0100          /* Bit that indicates response  */

#define RF_MSG_RREQ     0x0001          /* Read Request and response    */
#define RF_MSG_RRES     (RF_MSG_RREQ | RF_MSG_RESPONSE)

#define RF_MSG_WREQ     0x0002          /* Write Request and response   */
#define RF_MSG_WRES     (RF_MSG_WREQ | RF_MSG_RESPONSE)

#define RF_MSG_OREQ     0x0003          /* Open request and response    */
#define RF_MSG_ORES     (RF_MSG_OREQ | RF_MSG_RESPONSE)

#define RF_MSG_DREQ     0x0004          /* Delete request and response  */
#define RF_MSG_DRES     (RF_MSG_DREQ | RF_MSG_RESPONSE)

#define RF_MSG_TREQ     0x0005          /* Truncate request & response  */
#define RF_MSG_TRES     (RF_MSG_TREQ | RF_MSG_RESPONSE)

#define RF_MSG_SREQ     0x0006          /* Size request and response    */
#define RF_MSG_SRES     (RF_MSG_SREQ | RF_MSG_RESPONSE)

#define RF_MSG_MREQ     0x0007          /* Mkdir request and response   */
#define RF_MSG_MRES     (RF_MSG_MREQ | RF_MSG_RESPONSE)

#define RF_MSG_XREQ     0x0008          /* Rmdir request and response   */
#define RF_MSG_XRES     (RF_MSG_XREQ | RF_MSG_RESPONSE)

#define RF_MIN_REQ      RF_MSG_RREQ     /* Minimum request type         */
#define RF_MAX_REQ      RF_MSG_XREQ     /* Maximum request type         */

/* Message header fields present in each message */

#define RF_MSG_HDR                      /* Common message fields        */\
        uint16  rf_type;                /* Message type                 */\
        uint16  rf_status;              /* 0 in req, status in response */\
        uint32  rf_seq;                 /* Message sequence number      */\
        char    rf_name[RF_NAMLEN];     /* Null-terminated file name    */

/* The standard header present in all messages with no extra fields     */

/************************************************************************/
/*                                                                      */
/*                              Header                                  */
/*                                                                      */
/************************************************************************/
```

```
#pragma pack(2)
struct  rf_msg_hdr {                    /* Header fields present in each*/
        RF_MSG_HDR                      /*   remote file system message */
};
#pragma pack()

/************************************************************************/
/*                                                                      */
/*                              Read                                    */
/*                                                                      */
/************************************************************************/

#pragma pack(2)
struct  rf_msg_rreq     {               /* Remote file read request     */
        RF_MSG_HDR                      /* Header fields                */
        uint32  rf_pos;                 /* Position in file to read     */
        uint32  rf_len;                 /* Number of bytes to read      */
                                        /*   (between 1 and 1024)       */
};
#pragma pack()

#pragma pack(2)
struct  rf_msg_rres     {               /* Remote file read reply       */
        RF_MSG_HDR                      /* Header fields                */
        uint32  rf_pos;                 /* Position in file             */
        uint32  rf_len;                 /* Number of bytes that follow  */
                                        /*   (0 for EOF)                */
        char    rf_data[RF_DATALEN];    /* Array containing data from   */
                                        /*   the file                   */
};
#pragma pack()

/************************************************************************/
/*                                                                      */
/*                              Write                                   */
/*                                                                      */
/************************************************************************/

#pragma pack(2)
struct  rf_msg_wreq     {               /* Remote file write request    */
        RF_MSG_HDR                      /* Header fields                */
        uint32  rf_pos;                 /* Position in file             */
        uint32  rf_len;                 /* Number of valid bytes in     */
                                        /*   array that follows         */
```

```
        char    rf_data[RF_DATALEN];    /* Array containing data to be  */
                                        /*   written to the file        */
};
#pragma pack()

#pragma pack(2)
struct  rf_msg_wres     {               /* Remote file write response   */
        RF_MSG_HDR                      /* Header fields                */
        uint32  rf_pos;                 /* Original position in file    */
        uint32  rf_len;                 /* Number of bytes written      */
};
#pragma pack()

/************************************************************************/
/*                                                                      */
/*                              Open                                    */
/*                                                                      */
/************************************************************************/

#pragma pack(2)
struct  rf_msg_oreq     {               /* Remote file open request     */
        RF_MSG_HDR                      /* Header fields                */
        int32   rf_mode;                /* Xinu mode bits               */
};
#pragma pack()

#pragma pack(2)
struct  rf_msg_ores     {               /* Remote file open response    */
        RF_MSG_HDR                      /* Header fields                */
        int32   rf_mode;                /* Xinu mode bits               */
};
#pragma pack()

/************************************************************************/
/*                                                                      */
/*                              Size                                    */
/*                                                                      */
/************************************************************************/

#pragma pack(2)
struct  rf_msg_sreq     {               /* Remote file size request     */
        RF_MSG_HDR                      /* Header fields                */
};
#pragma pack()
```

```
#pragma pack(2)
struct  rf_msg_sres     {               /* Remote file status response  */
        RF_MSG_HDR                      /* Header fields                */
        uint32  rf_size;                /* Size of file in bytes        */
};
#pragma pack()

/************************************************************************/
/*                                                                      */
/*                             Delete                                   */
/*                                                                      */
/************************************************************************/

#pragma pack(2)
struct  rf_msg_dreq     {               /* Remote file delete request   */
        RF_MSG_HDR                      /* Header fields                */
};
#pragma pack()

#pragma pack(2)
struct  rf_msg_dres     {               /* Remote file delete response  */
        RF_MSG_HDR                      /* Header fields                */
};
#pragma pack()

/************************************************************************/
/*                                                                      */
/*                             Truncate                                 */
/*                                                                      */
/************************************************************************/

#pragma pack(2)
struct  rf_msg_treq     {               /* Remote file truncate request */
        RF_MSG_HDR                      /* Header fields                */
};
#pragma pack()

#pragma pack(2)
struct  rf_msg_tres     {               /* Remote file truncate response*/
        RF_MSG_HDR                      /* Header fields                */
};
#pragma pack()

/************************************************************************/
/*                                                                      */
```

```
/*                              Mkdir                                   */
/*                                                                      */
/************************************************************************/

#pragma pack(2)
struct  rf_msg_mreq     {               /* Remote file mkdir request    */
        RF_MSG_HDR                      /* Header fields                */
};
#pragma pack()

#pragma pack(2)
struct  rf_msg_mres     {               /* Remote file mkdir response   */
        RF_MSG_HDR                      /* Header fields                */
};
#pragma pack()

/************************************************************************/
/*                                                                      */
/*                              Rmdir                                   */
/*                                                                      */
/************************************************************************/

#pragma pack(2)
struct  rf_msg_xreq     {               /* Remote file rmdir request    */
        RF_MSG_HDR                      /* Header fields                */
};
#pragma pack()

#pragma pack(2)
struct  rf_msg_xres     {               /* Remote file rmdir response   */
        RF_MSG_HDR                      /* Header fields                */
};
#pragma pack()
```

In the file, constants that begin *RF_MSG_* define a unique type value for each message. For example, *RF_MSG_RREQ* defines the type value used in a *read request* message, and *RF_MSG_RRES* defines the type value used in a *read response* message. The implementation uses a trick to improve efficiency: rather than define arbitrary integers, the type of a response is formed by a logical or of the request type and constant *RF_MSG_RESPONSE*, which is defined to be 0x0100. That is, a response has the same type value as a request except that the low-order bit of the second byte is turned on.

The size of a message depends on the type. Many of the messages only need fields in the common header. For example, a file deletion request only requires a type (to indicate that it is a deletion request), a file name, and a sequence number. Thus, the

struct that defines a deletion request, *rf_msg_dreq*, only contains header fields. However, a *write request* message must include a file offset, the number of data bytes in the request, and the data to be written. Consequently, the struct that defines a *write request* message, *rf_msg_wreq*, includes three additional fields beyond the common header.

20.5 Remote File Server Communication (rfscomm)

Our remote file system software follows a principle that works well in many cases: the functionality is separated into two levels of software. A lower level handles details of communication with the remote server — it sends a message, waits for a response, and handles retransmission, if necessary. An upper level handles message semantics — it forms a message, passes the message to the lower level for transmission, receives a response, and interprets the response. The important idea is that because it only handles transmission and reception, the lower level does not need to understand or interpret the contents of a message. Consequently, a single function provides all lower-level functionality.

Examining the code will clarify the idea. Function *rfscomm* performs the action of sending a message to the remote file server and receiving a response. File *rfscomm.c* contains the code:

```
/* rfscomm.c - rfscomm */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rfscomm  -  Handle communication with RFS server (send request and
 *              receive a reply, including sequencing and retries)
 *------------------------------------------------------------------------
 */
int32   rfscomm (
         struct rf_msg_hdr *msg,        /* Message to send              */
         int32  mlen,                   /* Message length               */
         struct rf_msg_hdr *reply,      /* Buffer for reply             */
         int32  rlen                    /* Size of reply buffer         */
        )
{
        int32   i;                      /* Counts retries               */
        int32   retval;                 /* Return value                 */
        int32   seq;                    /* Sequence for this exchange   */
        int16   rtype;                  /* Reply type in host byte order*/
        int32   slot;                   /* UDP slot                     */

        /* For the first time after reboot, register the server port */
```

```
if ( ! Rf_data.rf_registered ) {
        if ( (retval = udp_register(Rf_data.rf_ser_ip,
                        Rf_data.rf_ser_port,
                        Rf_data.rf_loc_port)) == SYSERR) {
                return SYSERR;
        }
        Rf_data.rf_udp_slot = retval;
        Rf_data.rf_registered = TRUE;
}

/* Assign message next sequence number */

seq = Rf_data.rf_seq++;
msg->rf_seq = htonl(seq);

/* Repeat RF_RETRIES times: send message and receive reply */

for (i=0; i<RF_RETRIES; i++) {

        /* Send a copy of the message */

        retval = udp_send(Rf_data.rf_udp_slot, (char *)msg,
                mlen);
        if (retval == SYSERR) {
                kprintf("Cannot send to remote file server\n");
                return SYSERR;
        }

        /* Receive a reply */

        retval = udp_recv(Rf_data.rf_udp_slot, (char *)reply,
                rlen, RF_TIMEOUT);

        if (retval == TIMEOUT) {
                continue;
        } else if (retval == SYSERR) {
                kprintf("Error reading remote file reply\n");
                return SYSERR;
        }

        /* Verify that sequence in reply matches request */

        if (ntohl(reply->rf_seq) != seq) {
                continue;
        }
```

```
                /* Verify the type in the reply matches the request */

                rtype = ntohs(reply->rf_type);
                if (rtype != ( ntohs(msg->rf_type) | RF_MSG_RESPONSE) ) {
                        continue;
                }

                return retval;          /* Return length to caller */
        }

        /* Retries exhausted without success */

        kprintf("Timeout on exchange with remote file server\n");
        return TIMEOUT;
}
```

The four arguments specify the address of a message that should be sent to the server, the length of the message, the address of a buffer that will hold a response message, and the length of the buffer. After assigning a unique sequence number to the message, *rfscomm* enters a loop that iterates *RF_RETRIES* times. On each iteration, *rfscomm* uses function *udp_send* to send a copy of the request message over the network† and function *udp_recv* to receive a response.

Udp_recv allows a caller to specify a maximum time to wait for a response; *rfscomm* specifies *RF_TIMEOUT*.‡ If no message arrives within the specified time, *udp_recv* returns the value *TIMEOUT*, and the loop continues by transmitting another copy of the request. If no response arrives after *RF_RETRIES* attempts, *rfscomm* returns *TIMEOUT* to its caller.

If a response does arrive, *rfscomm* verifies that the sequence number matches the sequence number in the outgoing request and the message type in the incoming message is the response for the outgoing request. If either test fails, the server formed the message incorrectly or the message was intended for another client on the network. In either case, *rfscomm* continues the loop, sending another copy of the request and waiting for a response to arrive. If the two tests succeed, the incoming message is a valid response, and *rfscomm* returns the length of the response to its caller.

20.6 Sending A Basic Message (rfsndmsg)

To understand how *rfscomm* functions, consider a message that only requires the common header fields. For example, the request and response messages used for a *truncate* operation consist of a message header. Because multiple message types only have the common header fields, function *rfsndmsg* has been created to send such a message. File *rfsndmsg.c* contains the code.

---

†We say that *rfscomm* sends a copy of the message because the original message remains unchanged.

‡*RF_TIMEOUT* is defined to be 1000 milliseconds (i.e., one second), which is ample time for a client to transmit a message across a network to a server and a server to send a response back to the client.

```
/* rfsndmsg.c - rfsndmsg */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rfsndmsg  -  Create and send a message that only has header fields
 *------------------------------------------------------------------------
 */
status  rfsndmsg (
         uint16 type,                   /* Message type                 */
         char   *name                   /* Null-terminated file name    */
        )
{
        struct  rf_msg_hdr  req;        /* Request message to send      */
        struct  rf_msg_hdr resp;        /* Buffer for response          */
        int32   retval;                 /* Return value                 */
        char    *to;                    /* Used during name copy        */

        /* Form a request */

        req.rf_type = htons(type);
        req.rf_status = htons(0);
        req.rf_seq = 0;                 /* Rfscomm will set sequence    */
        to = req.rf_name;
        while ( (*to++ = *name++) ) {   /* Copy name to request         */
                ;
        }

        /* Send message and receive response */

        retval = rfscomm(&req,  sizeof(struct rf_msg_hdr),
                         &resp, sizeof(struct rf_msg_hdr) );

        /* Check response */

        if (retval == SYSERR) {
                return SYSERR;
        } else if (retval == TIMEOUT) {
                kprintf("Timeout during remote file server access\n");
                return SYSERR;
        } else if (ntohl(resp.rf_status) != 0) {
                return SYSERR;
        }

        return OK;
}
```

Rfsndmsg takes two arguments that specify the type of the message to send and the name of a file. To create a request message, the code assigns a value to each field of variable *req*. It then calls *rfscomm* to transmit the message and receive a response. If *rfscomm* reports an error or timeout or if the status in the response indicates an error, *rfsndmsg* returns *SYSERR* to its caller. Otherwise, *rfsndmsg* returns *OK*.

20.7 Network Byte Order

Remote file access raises an important consideration: the format of integers (i.e., endianness) depends on the computer architecture. If one were to transfer an integer from the memory of one computer directly to the memory on another, the numeric value of the integer on the second computer may differ from the numeric value on the first. To accommodate differences, software that sends data over a computer network follows the convention of converting integers from the local byte order to a standard known as *network byte order*, and software that receives data from a computer network converts integers from network byte order to the local byte order. We can summarize:

> *To accommodate differences in endianness, an integer value sent from one computer to another is converted to network byte order before sending and converted to local byte order upon reception. In our design, upper-level functions perform the conversion.*

Xinu follows the Unix naming convention for byte-order transform functions. Function *htonl* (*htons*) transforms an integer (a short integer) from local host byte order to network byte order; function *ntohl* (*ntohs*) transforms an integer (a short integer) from network byte order to local byte order. For example, function *rfsndmsg* uses *htons* to convert the integers that specify the message type and status from local byte order to network byte order.

20.8 A Remote File System Using A Device Paradigm

As we have seen, Xinu uses a device paradigm for both devices and files. The remote file system follows the pattern. Figure 20.1 shows an excerpt from the Xinu *Configuration* file that defines the type of a remote file system master device and a set of remote file pseudo-devices.

```
/* Remote File System master device type */

rfs:
        on udp
                -i rfsinit      -o rfsopen      -c ioerr
                -r ioerr        -g ioerr        -p ioerr
                -w ioerr        -s ioerr        -n rfscontrol
                -intr NULL

/* Remote file pseudo-device type */

rfl:
        on rfs
                -i rflinit      -o ioerr        -c rflclose
                -r rflread      -g rflgetc      -p rflputc
                -w rflwrite     -s rflseek      -n ioerr
                -intr NULL
```

Figure 20.1 Excerpt from a Xinu Configuration file that defines the two device types used by the remote file system.

Figure 20.2 contains an excerpt from the Configuration file that defines a remote file system master device (*RFILESYS*) and a set of six remote file pseudo-devices (*RFILE0* through *RFILE5*).

```
/* Remote file system master device (one per system) */

                RFILESYS is rfs on udp

/* Remote file pseudo-devices (many instances per system) */

                RFILE0 is rfl on rfs
                RFILE1 is rfl on rfs
                RFILE2 is rfl on rfs
                RFILE3 is rfl on rfs
                RFILE4 is rfl on rfs
                RFILE5 is rfl on rfs
```

Figure 20.2 Excerpt from a Xinu Configuration file that defines devices used by the remote file system.

When an application calls *open* on the remote file system master device, the call allocates one of the remote file pseudo-devices and returns the device ID of the allocat-

ed pseudo-device. The application uses the device ID in calls to *read* and *write*, and eventually calls *close* to deallocate the pseudo-device. The next sections define the device driver functions used for both the remote file system master device and the remote file pseudo-devices.

20.9 Opening A Remote File (rfsopen)

To open a remote file, a program calls *open* on device *RFILESYS*, supplying a file name and mode argument. *Open* invokes function *rfsopen*, which forms a request and uses *rfscomm* to communicate with the remote file server. If it succeeds, the call to *open* returns the descriptor of a remote file pseudo-device that is associated with the open file (i.e., can be used to write data into the file or read data from the file). File *rfsopen.c* contains the code:

```
/* rfsopen.c - rfsopen */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rfsopen  -  Allocate a remote file pseudo-device for a specific file
 *------------------------------------------------------------------------
 */

devcall rfsopen (
         struct dentry  *devptr,        /* Entry in device switch table */
         char   *name,                  /* File name to use             */
         char   *mode                   /* Mode chars: 'r' 'w' 'o' 'n'  */
        )
{
        struct  rflcblk *rfptr;         /* Ptr to control block entry   */
        struct  rf_msg_oreq msg;        /* Message to be sent           */
        struct  rf_msg_ores resp;       /* Buffer to hold response      */
        int32   retval;                 /* Return value from rfscomm    */
        int32   len;                    /* Counts chars in name         */
        char    *nptr;                  /* Pointer into name string     */
        char    *fptr;                  /* Pointer into file name       */
        int32   i;                      /* General loop index           */

        /* Wait for exclusive access */

        wait(Rf_data.rf_mutex);

        /* Search control block array to find a free entry */
```

```
for(i=0; i<Nrfl; i++) {
        rfptr = &rfltab[i];
        if (rfptr->rfstate == RF_FREE) {
                break;
        }
}
if (i >= Nrfl) {                /* No free table slots remain   */
        signal(Rf_data.rf_mutex);
        return SYSERR;
}

/* Copy name into free table slot */

nptr = name;
fptr = rfptr->rfname;
len = 0;
while ( (*fptr++ = *nptr++) != NULLCH) {
        len++;
        if (len >= RF_NAMLEN) { /* File name is too long        */
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }
}

/* Verify that name is non-null */

if (len==0) {
        signal(Rf_data.rf_mutex);
        return SYSERR;
}

/* Parse mode string */

if ( (rfptr->rfmode = rfsgetmode(mode)) == SYSERR ) {
        signal(Rf_data.rf_mutex);
        return SYSERR;
}

/* Form an open request to create a new file or open an old one */

msg.rf_type = htons(RF_MSG_OREQ);/* Request a file open         */
msg.rf_status = htons(0);
msg.rf_seq = 0;                 /* Rfscomm fills in seq. number */
nptr = msg.rf_name;
memset(nptr, NULLCH, RF_NAMLEN);/* Initialize name to zero bytes*/
```

```
        while ( (*nptr++ = *name++) != NULLCH ) { /* Copy name to req.  */
                ;
        }
        msg.rf_mode = htonl(rfptr->rfmode); /* Set mode in request      */

        /* Send message and receive response */

        retval = rfscomm((struct rf_msg_hdr *)&msg,
                                        sizeof(struct rf_msg_oreq),
                        (struct rf_msg_hdr *)&resp,
                                        sizeof(struct rf_msg_ores) );

        /* Check response */

        if (retval == SYSERR) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        } else if (retval == TIMEOUT) {
                kprintf("Timeout during remote file open\n\r");
                signal(Rf_data.rf_mutex);
                return SYSERR;
        } else if (ntohs(resp.rf_status) != 0) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }

        /* Set initial file position */

        rfptr->rfpos = 0;

        /* Mark state as currently used */

        rfptr->rfstate = RF_USED;

        /* Return device descriptor of newly created pseudo-device */

        signal(Rf_data.rf_mutex);
        return rfptr->rfdev;
}
```

Before proceeding to check its arguments, *rfsopen* checks the remote pseudo-devices to ensure that one is available. The code then checks the file name to ensure that the name is less than the maximum allowed and the mode string to ensure that the specification is valid.

Before it allocates the remote file device, *rfsopen* must communicate with the remote server to ensure the remote server agrees the file can be opened. The code creates a request message, and uses *rfscomm* to send the message to the server. If a positive response arrives, *rfsopen* marks the control block entry for the remote file device as being used, sets the initial file position to zero, and returns the descriptor to the caller.

20.10 Checking The File Mode (rfsgetmode)

When it needs to check the file mode argument, *rfsopen* calls function *rfsgetmode*, passing the mode string as an argument. The code can be found in file *rfsgetmode.c*:

```
/* rfsgetmode.c - rfsgetmode */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rfsgetmode  -  Parse mode argument and generate integer of mode bits
 *------------------------------------------------------------------------
 */

int32   rfsgetmode (
         char   *mode                   /* String of mode characters    */
        )
{
        int32   mbits;                  /* Mode bits to return (in host */
                                        /*    byte order)               */
        char    ch;                     /* Next character in mode string*/

        mbits = 0;

        /* Mode string specifies:                                       */
        /*      r - read                                                */
        /*      w - write                                               */
        /*      o - old (file must exist)                               */
        /*      n - new (create a new file)                             */

        while ( (ch = *mode++) != NULLCH) {
                switch (ch) {

                    case 'r':   if (mbits&RF_MODE_R) {
                                        return SYSERR;
                                }
                                mbits |= RF_MODE_R;
                                continue;
```

```
                case 'w':   if (mbits&RF_MODE_W) {
                                    return SYSERR;
                            }
                            mbits |= RF_MODE_W;
                            continue;

                case 'o':   if (mbits&RF_MODE_O || mbits&RF_MODE_N) {
                                    return SYSERR;
                            }
                            mbits |= RF_MODE_O;
                            break;

                case 'n':   if (mbits&RF_MODE_O || mbits&RF_MODE_N) {
                                    return SYSERR;
                            }
                            mbits |= RF_MODE_N;
                            break;

                default:    return SYSERR;
            }
        }

        /* If neither read nor write specified, allow both */

        if ( (mbits&RF_MODE_RW) == 0 ) {
                mbits |= RF_MODE_RW;
        }
        return mbits;
}
```

Rfsgetmode extracts characters from the mode string, ensures each is valid, and checks for illegal combinations (e.g., a mode string cannot specify both *new* and *old* modes). As it parses the mode string, *rfsgetmode* sets the bits in integer *mbits*. Once it has finished examining the string and checking the combinations, *rfsgetmode* returns integer *mbits* to the caller.

20.11 Closing A Remote File (rflclose)

Once a process has finished using a file, the process can call *close* to release the remote file device and make it available for the system to use for another file. For a remote file device, *close* invokes *rflclose*. In our implementation, closing a remote file is trivial. Function *rflclose.c* contains the code:

```
/* rflclose.c - rflclose */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rflclose  -  Close a remote file device
 *------------------------------------------------------------------------
 */
devcall rflclose (
          struct dentry *devptr         /* Entry in device switch table */
        )
{
        struct  rflcblk *rfptr;         /* Pointer to control block     */

        /* Wait for exclusive access */

        wait(Rf_data.rf_mutex);

        /* Verify remote file device is open */

        rfptr = &rfltab[devptr->dvminor];
        if (rfptr->rfstate == RF_FREE) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }

        /* Mark device closed */

        rfptr->rfstate = RF_FREE;
        signal(Rf_data.rf_mutex);
        return OK;
}
```

After verifying that the device is currently open, *rflclose* sets the state of the control block entry to *RF_FREE*. Note that this version of *rflclose* does not inform the remote file server that the file is closed. The exercises suggest redesigning the system to inform the remote server when a file is closed.

20.12 Reading From A Remote File (rflread)

Once a remote file has been opened, a process can read data from the file. Driver function *rflread* performs the *read* operation. The code can be found in file *rflread.c*:

```
/* rflread.c - rflread */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rflread  -  Read data from a remote file
 *------------------------------------------------------------------------
 */
devcall rflread (
          struct dentry *devptr,        /* Entry in device switch table */
          char  *buff,                  /* Buffer of bytes              */
          int32 count                   /* Count of bytes to read       */
        )
{
        struct  rflcblk *rfptr;         /* Pointer to control block     */
        int32   retval;                 /* Return value                 */
        struct  rf_msg_rreq  msg;       /* Request message to send      */
        struct  rf_msg_rres resp;       /* Buffer for response          */
        int32   i;                      /* Counts bytes copied          */
        char    *from, *to;             /* Used during name copy        */
        int32   len;                    /* Length of name               */

        /* Wait for exclusive access */

        wait(Rf_data.rf_mutex);

        /* Verify count is legitimate */

        if ( (count <= 0) || (count > RF_DATALEN) ) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }

        /* Verify pseudo-device is in use */

        rfptr = &rfltab[devptr->dvminor];

        /* If device not currently in use, report an error */

        if (rfptr->rfstate == RF_FREE) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }

        /* Verify pseudo-device allows reading */
```

```
if ((rfptr->rfmode & RF_MODE_R) == 0) {
        signal(Rf_data.rf_mutex);
        return SYSERR;
}

/* Form read request */

msg.rf_type = htons(RF_MSG_RREQ);
msg.rf_status = htons(0);
msg.rf_seq = 0;                 /* Rfscomm will set sequence    */
from = rfptr->rfname;
to = msg.rf_name;
memset(to, NULLCH, RF_NAMLEN);  /* Start name as all zero bytes */
len = 0;
while ( (*to++ = *from++) ) {   /* Copy name to request         */
        if (++len >= RF_NAMLEN) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }
}
msg.rf_pos = htonl(rfptr->rfpos);/* Set file position           */
msg.rf_len = htonl(count);      /* Set count of bytes to read   */

/* Send message and receive response */

retval = rfscomm((struct rf_msg_hdr *)&msg,
                                sizeof(struct rf_msg_rreq),
                 (struct rf_msg_hdr *)&resp,
                                sizeof(struct rf_msg_rres) );

/* Check response */

if (retval == SYSERR) {
        signal(Rf_data.rf_mutex);
        return SYSERR;
} else if (retval == TIMEOUT) {
        kprintf("Timeout during remote file read\n");
        signal(Rf_data.rf_mutex);
        return SYSERR;
} else if (ntohs(resp.rf_status) != 0) {
        signal(Rf_data.rf_mutex);
        return SYSERR;
}

/* Copy data to application buffer and update file position */
```

```
        for (i=0; i<htonl(resp.rf_len); i++) {
                *buff++ = resp.rf_data[i];
        }
        rfptr->rfpos += htonl(resp.rf_len);

        signal(Rf_data.rf_mutex);
        return htonl(resp.rf_len);
}
```

Rflread begins by checking argument *count* to verify that the request is in range. It then verifies that the pseudo-device has been opened and the mode allows reading. Once the checking is complete, *rflread* performs the *read* operation: it forms a message, uses *rfscomm* to transmit a copy to the server and receive a response, and interprets the response.

If *rfscomm* returns a valid response, the message will include the data that has been read. *Rflread* copies the data from the response message into the caller's buffer, updates the file position, and returns the number of bytes to the caller.

20.13 Writing To A Remote File (rflwrite)

Writing to a remote file follows the same general paradigm as reading from a remote file. Driver function *rflwrite* performs the *write* operation; the code can be found in file *rflwrite.c*:

```
/* rflwrite.c - rflwrite */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rflwrite  -  Write data to a remote file
 *------------------------------------------------------------------------
 */
devcall rflwrite (
          struct dentry *devptr,        /* Entry in device switch table */
          char  *buff,                  /* Buffer of bytes              */
          int32 count                   /* Count of bytes to write      */
        )
{
        struct  rflcblk *rfptr;         /* Pointer to control block     */
        int32   retval;                 /* Return value                 */
        struct  rf_msg_wreq  msg;       /* Request message to send      */
        struct  rf_msg_wres resp;       /* Buffer for response          */
        char    *from, *to;             /* Used to copy name            */
```

```
int     i;                      /* Counts bytes copied into req */
int32   len;                    /* Length of name               */

/* Wait for exclusive access */

wait(Rf_data.rf_mutex);

/* Verify count is legitimate */

if ( (count <= 0) || (count > RF_DATALEN) ) {
        signal(Rf_data.rf_mutex);
        return SYSERR;
}

/* Verify pseudo-device is in use and mode allows writing */

rfptr = &rfltab[devptr->dvminor];
if ( (rfptr->rfstate == RF_FREE) ||
     ! (rfptr->rfmode & RF_MODE_W) ) {
        signal(Rf_data.rf_mutex);
        return SYSERR;
}

/* Form write request */

msg.rf_type = htons(RF_MSG_WREQ);
msg.rf_status = htons(0);
msg.rf_seq = 0;                 /* Rfscomm will set sequence    */
from = rfptr->rfname;
to = msg.rf_name;
memset(to, NULLCH, RF_NAMLEN);  /* Start name as all zero bytes */
len = 0;
while ( (*to++ = *from++) ) {   /* Copy name to request         */
        if (++len >= RF_NAMLEN) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }
}
while ( (*to++ = *from++) ) {   /* Copy name into request       */
        ;
}
msg.rf_pos = htonl(rfptr->rfpos);/* Set file position           */
msg.rf_len = htonl(count);      /* Set count of bytes to write  */
for (i=0; i<count; i++) {       /* Copy data into message       */
        msg.rf_data[i] = *buff++;
```

```
        }
        while (i < RF_DATALEN) {
                msg.rf_data[i++] = NULLCH;
        }

        /* Send message and receive response */

        retval = rfscomm((struct rf_msg_hdr *)&msg,
                                        sizeof(struct rf_msg_wreq),
                        (struct rf_msg_hdr *)&resp,
                                sizeof(struct rf_msg_wres) );

        /* Check response */

        if (retval == SYSERR) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        } else if (retval == TIMEOUT) {
                kprintf("Timeout during remote file read\n");
                signal(Rf_data.rf_mutex);
                return SYSERR;
        } else if (ntohs(resp.rf_status) != 0) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }

        /* Report results to caller */

        rfptr->rfpos += ntohl(resp.rf_len);

        signal(Rf_data.rf_mutex);
        return ntohl(resp.rf_len);
}
```

As with a *read* operation, *rflwrite* begins by checking the *count* argument, verifying that the pseudo-device is open and the mode allows writing. *Rflwrite* then forms a request message and uses *rfscomm* to send the message to the server.

Unlike a *read* request, an outgoing *write* request contains data. Thus, when forming the request, *rflwrite* copies data from the user's buffer into the request message. When a response arrives, the response message does not contain a copy of the data that has been written. Thus, *rflwrite* uses the status field in the message to determine whether to report success or failure to the caller.

20.14 Seeking On A Remote File (rflseek)

How should a *seek* operation be implemented for our remote file system? There are two possibilities. In one design, the system sends a message to the remote server, and the remote server seeks to the specified location in the file. In the other design, all location data is kept on the local computer, and each request to the server contains an explicit file position.

Our implementation uses the latter: the current file position is stored in the control block entry for a remote file device. When *read* is called, *rflread* requests data from the server and updates the file position in the control block entry accordingly. The remote server does not record a position because each request includes explicit position information. An exercise considers a consequence of the design.

Because all file position information is stored on the client, a *seek* operation can be performed locally. That is, the software stores the file position in the control block entry for use on the next *read* or *write* operation. Function *rflseek* performs the *seek* operation on a remote file device. The code can be found in file *rflseek.c*:

```
/* rflseek.c - rflseek */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rflseek  -  Change the current position in an open file
 *------------------------------------------------------------------------
 */
devcall rflseek (
          struct dentry *devptr,        /* Entry in device switch table */
          uint32 pos                    /* New file position            */
        )
{
        struct  rflcblk *rfptr;         /* Pointer to control block     */

        /* Wait for exclusive access */

        wait(Rf_data.rf_mutex);

        /* Verify remote file device is open */

        rfptr = &rfltab[devptr->dvminor];
        if (rfptr->rfstate == RF_FREE) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }
```

```
        /* Set the new position */

        rfptr->rfpos = pos;
        signal(Rf_data.rf_mutex);
        return OK;
}
```

The code is trivial. After obtaining exclusive access, *rflseek* verifies that the device has been opened. It then stores the file position argument in field *rfpos* of the control block, signals the mutual exclusion semaphore, and returns. There is no need to contact the remote server.

20.15 Character I/O On A Remote File (rflgetc, rflputc)

Using a remote file server to read and write individual bytes of data is expensive because a message must be sent to the server for each byte. Rather than prohibit character I/O, our implementations of *getc* and *putc*, call *rflread* and *rflwrite*, respectively. Thus, we allow a programmer to decide whether the cost is reasonable. Files *rflgetc.c* and *rflputc.c* contain the code.

```
/* rflgetc.c - rflgetc */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  rflgetc  -  Read one character from a remote file
 *------------------------------------------------------------------------
 */
devcall rflgetc(
        struct  dentry  *devptr         /* Entry in device switch table */
        )
{
        char    ch;                     /* Character to read            */
        int32   retval;                 /* Return value                 */

        retval = rflread(devptr, &ch, 1);

        if (retval != 1) {
                return SYSERR;
        }

        return (devcall)ch;
}
```

```
/* rflputc.c - rflputc */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  rflputc  -  Write one character to a remote file
 *------------------------------------------------------------------------
 */
devcall rflputc(
        struct  dentry  *devptr,        /* Entry in device switch table */
        char    ch                      /* Character to write           */
        )
{
        struct  rflcblk *rfptr;         /* Pointer to rfl control block */

        rfptr = &rfltab[devptr->dvminor];

        if (rflwrite(devptr, &ch, 1) != 1) {
                return SYSERR;
        }

        return OK;
}
```

20.16 Remote File System Control Functions (rfscontrol)

Several file operations are needed beyond *open*, *read*, *write*, and *close*. For example, it may be necessary to delete a file. The Xinu remote file system uses the *control* function to implement such functions. The table in Figure 20.3 lists the set of symbolic constants used for control functions along with the meaning of each.

Constant	Meaning
RFS_CTL_DEL	Delete the named file
RFS_CTL_TRUNC	Truncate a named file to zero bytes
RFS_CTL_MKDIR	Make a directory
RFS_CTL_RMDIR	Remove a directory
RFS_CTL_SIZE	Return the current size of a file in bytes

Figure 20.3 Control functions used with the remote file system.

Because they refer to operations outside individual files, *control* operations are performed on the master device, *RFILESYS*, rather than on an individual file pseudo-device. Driver function *rfscontrol* implements the *control* operation; the code can be found in file *rfscontrol.c*:

```
/* rfscontrol.c - rfscontrol */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rfscontrol  -  Provide control functions for the remote file system
 *------------------------------------------------------------------------
 */
devcall rfscontrol (
         struct dentry  *devptr,        /* Entry in device switch table */
         int32  func,                   /* A control function           */
         int32  arg1,                   /* Argument #1                  */
         int32  arg2                    /* Argument #2                  */
        )
{
        int32   len;                    /* Length of name               */
        struct  rf_msg_sreq msg;        /* Buffer for size request      */
        struct  rf_msg_sres resp;       /* Buffer for size response     */
        struct  rflcblk *rfptr;         /* Pointer to entry in rfltab   */
        char    *to, *from;             /* Used during name copy        */
        int32   retval;                 /* Return value                 */

        /* Wait for exclusive access */

        wait(Rf_data.rf_mutex);

        /* Check length and copy (needed for size) */

        rfptr = &rfltab[devptr->dvminor];
        from = rfptr->rfname;
        to = msg.rf_name;
        len = 0;
        memset(to, NULLCH, RF_NAMLEN);  /* Start name as all zeroes     */
        while ( (*to++ = *from++) ) {   /* Copy name to message         */
                len++;
                if (len >= (RF_NAMLEN - 1) ) {
                        signal(Rf_data.rf_mutex);
                        return SYSERR;
                }
        }
```

```
switch (func) {

/* Delete a file */

case RFS_CTL_DEL:
        if (rfsndmsg(RF_MSG_DREQ, (char *)arg1) == SYSERR) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }
        break;

/* Truncate a file */

case RFS_CTL_TRUNC:
        if (rfsndmsg(RF_MSG_TREQ, (char *)arg1) == SYSERR) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }
        break;



/* Make a directory */

case RFS_CTL_MKDIR:
        if (rfsndmsg(RF_MSG_MREQ, (char *)arg1) == SYSERR) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }
        break;

/* Remove a directory */

case RFS_CTL_RMDIR:
        if (rfsndmsg(RF_MSG_XREQ, (char *)arg1) == SYSERR) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }
        break;

/* Obtain current file size (non-standard message size) */

case RFS_CTL_SIZE:

        /* Hand-craft a size request message */
```

```
                msg.rf_type = htons(RF_MSG_SREQ);
                msg.rf_status = htons(0);
                msg.rf_seq = 0;         /* Rfscomm will set the seq num */

                /* Send the request to server and obtain a response     */

                retval = rfscomm( (struct rf_msg_hdr *)&msg,
                                        sizeof(struct rf_msg_sreq),
                                  (struct rf_msg_hdr *)&resp,
                                        sizeof(struct rf_msg_sres) );
                if ( (retval == SYSERR) || (retval == TIMEOUT) ) {
                        signal(Rf_data.rf_mutex);
                        return SYSERR;
                } else {
                        signal(Rf_data.rf_mutex);
                        return ntohl(resp.rf_size);
                }

        default:
                kprintf("rfscontrol: function %d not valid\n", func);
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }

        signal(Rf_data.rf_mutex);
        return OK;
}
```

For all the control functions, argument *arg1* contains a pointer to a null-terminated file name. After it obtains exclusive access and checks the length of the file name, *rfscontrol* uses the function argument to choose among several cases that correspond to file deletion, file truncation, directory creation, directory deletion, or a file size request. In each case, *rfscontrol* must send a message to the remote server and receive a response.

Except for a file size request, all messages to the server only include the common header fields. Therefore, for all functions except a size request, *rfscontrol* uses function *rfsndmsg* to generate and send a request to the remote server. For a size request, *rfscontrol* creates a message in variable *msg*, and uses *rfscomm* to send the message and receive a response. To avoid scanning the file name twice, *rfscontrol* copies the file name into the name field of variable *msg* as it checks the length of the name. Thus, no extra copy is needed when *rfscontrol* creates a size request. If a valid response arrives to a size request, *rfscontrol* extracts the file size from the response, converts it to local byte order, and returns the size to the caller. In all other cases, *rfscontrol* returns a status of either *OK* or *SYSERR*.

20.17 Initializing The Remote File System (rfsinit, rflinit)

Because the design includes both a remote file system master device and a set of remote file pseudo-devices, two initialization functions are needed. The first, *rfsinit*, initializes the control block associated with the master device.

Data for the master device is kept in global variable *Rf_data*. *Rfsinit* fills in fields of the structure with the remote server's IP address and UDP port number. It also allocates a mutual exclusion semaphore and stores the semaphore ID in the structure. *Rfsinit* sets field *rf_registered* to *FALSE*, indicating that before communication with the server is possible, the UDP port of the server must be registered with the network code. File *rfsinit.c* contains the code:

```
/* rfsinit.c - rfsinit */

#include <xinu.h>

struct  rfdata  Rf_data;

/*------------------------------------------------------------------------
 *  rfsinit  -  Initialize the remote file system master device
 *------------------------------------------------------------------------
 */
devcall rfsinit(
          struct dentry *devptr         /* Entry in device switch table */
        )
{

        /* Choose an initial message sequence number */

        Rf_data.rf_seq = 1;

        /* Set the server IP address, server port, and local port */

        if ( dot2ip(RF_SERVER_IP, &Rf_data.rf_ser_ip) == SYSERR ) {
                panic("invalid IP address for remote file server");
        }
        Rf_data.rf_ser_port = RF_SERVER_PORT;
        Rf_data.rf_loc_port = RF_LOC_PORT;

        /* Create a mutual exclusion semaphore */

        if ( (Rf_data.rf_mutex = semcreate(1)) == SYSERR ) {
                panic("Cannot create remote file system semaphore");
        }
```

```
        /* Specify that the server port is not yet registered */

        Rf_data.rf_registered = FALSE;

        return OK;
}
```

Function *rflinit* handles initialization of individual remote file devices. The code can be found in file *rflinit.c*:

```
/* rflinit.c - rflinit */

#include <xinu.h>

struct  rflcblk rfltab[Nrfl];           /* Remote file control blocks   */

/*------------------------------------------------------------------------
 *  rflinit  -  Initialize a remote file device
 *------------------------------------------------------------------------
 */
devcall rflinit(
          struct dentry *devptr         /* Entry in device switch table */
        )
{
        struct  rflcblk *rflptr;        /* Ptr. to control block entry  */
        int32   i;                      /* Walks through name arrary    */

        rflptr = &rfltab[ devptr->dvminor ];

        /* Initialize entry to unused */

        rflptr->rfstate = RF_FREE;
        rflptr->rfdev = devptr->dvnum;
        for (i=0; i<RF_NAMLEN; i++) {
                rflptr->rfname[i] = NULLCH;
        }
        rflptr->rfpos = rflptr->rfmode = 0;
        return OK;
}
```

Rflinit sets the state of the entry to *RF_FREE* to indicate that the entry is currently unused. It also zeroes the name and mode fields. If the state is marked *RF_FREE*, no references should occur to other fields of the entry. Placing zeroes in the fields aids debugging.

20.18 Perspective

As with a local file system, the most complex decision involved in the design of a remote file system arises from the need to choose a balance between efficiency and sharing. To understand the choice, imagine multiple applications running on multiple computers all sharing a single file. At one extreme, to guarantee last-write semantics on a shared file, each file operation must be sent to the remote server so the requests can be serialized and operations can be applied to the file in the order they occur. At the other extreme, efficiency is maximized when a computer can cache files (or parts of files) and access items from the local cache. The goal is to devise a remote file system that maximizes performance when no sharing occurs, guarantees correctness in the presence of sharing, and transitions gracefully and automatically between the two extremes.

Sharing a remote file among multiple clients creates some unexpected subtleties. For example, consider the *seek* operation. Suppose a process on computer 1 writes 1000 bytes to a file. Then a process on computer 2 seeks to the end of the file and writes an additional 1000 bytes. Unless it contacts the server, the file access software on computer 1 cannot know that the file has been extended. Therefore, the software cannot determine whether a *seek* operation is valid without contacting the server. More important, the size of the file can continue to change while the communication takes place.

20.19 Summary

A remote file access mechanism allows applications running on a client computer to access files stored on a remote server. The example design uses a device paradigm in which an application calls *open* on the remote file system master device to obtain the ID of an individual remote file pseudo-device. The application can then use *read* and *write* on the pseudo-device.

When an application accesses a remote file, the remote file software creates a message, sends the message to the remote file server, waits for a response, and interprets the response. The software transmits each request multiple times in case the network drops a packet or the server is too busy to answer.

Operations such as file deletion, file truncation, creating and removing directories, and determining the current size of a file are handled with the *control* function. As with data transfer operations, each call to *control* results in the transmission of a request message and a response from the server.

EXERCISES

20.1 Modify the remote file server and *rflclose*. Arrange to have *rflclose* send a message to the server when a file is closed, and have the server send a response.

20.2 The underlying protocol limits a *read* request to *RF_DATALEN* bytes, and *rflread* rejects any call that specifies more. Modify *rflread* to allow a user to request an arbitrary size, but still limit the size in a request message to *RF_DATALEN* (i.e., don't reject large requests, but limit the data returned to *RF_DATALEN* bytes).

20.3 As an alternative to the exercise above, devise a system in which *rflread* permits a caller to specify an arbitrary size *read* and sends multiple requests to the server to satisfy the request.

20.4 The code in *rflgetc* calls *rflread* directly. What potential problem does such a direct call introduce? Modify the code to use the device switch table when making the call.

20.5 Our design keeps position information on the client side, which makes *seek* extremely efficient. What limitation does the design impose? Hint: consider shared files.

20.6 Consider an alternative design for the remote file system that improves efficiency. Arrange *rflread* so it always requests *RF_DATALEN* bytes, even if the caller requests fewer. Place extra bytes in a cache, making them available to subsequent calls.

20.7 In the previous exercise, what is the chief disadvantage of caching data for subsequent *reads*? Hint: consider shared access to the server.

20.8 Consider what happens if two clients attempt to use the remote file server at the same time. When the clients boot, they each start their packet sequence number at 1, which makes the probability of conflict high. Revise the system to use a random starting sequence number (and revise the server to accept arbitrary sequence numbers).

20.9 Redesign the remote file system to allow multiple clients to share a given file.

Chapter Contents

21.1 Introduction, 547
21.2 Transparency And A Namespace Abstraction, 547
21.3 Myriad Naming Schemes, 548
21.4 Naming System Design Alternatives, 550
21.5 Thinking About Names Syntactically, 550
21.6 Patterns And Replacements, 551
21.7 Prefix Patterns, 551
21.8 Implementation Of A Namespace, 552
21.9 Namespace Data Structures And Constants, 552
21.10 Adding Mappings To The Namespace Prefix Table, 553
21.11 Mapping Names With The Prefix Table, 555
21.12 Opening A Named File, 559
21.13 Namespace Initialization, 560
21.14 Ordering Entries In The Prefix Table, 562
21.15 Choosing A Logical Namespace, 563
21.16 A Default Hierarchy And The Null Prefix, 564
21.17 Additional Object Manipulation Functions, 564
21.18 Advantages And Limits Of The Namespace Approach, 566
21.19 Generalized Patterns, 566
21.20 Perspective, 567
21.21 Summary, 568

21

A Syntactic Namespace

A rose by any other name ...

— William Shakespeare

21.1 Introduction

Chapter 14 outlines a set of device-independent I/O operations, including *read* and *write*, and shows how a device switch table provides an efficient mapping between high-level operations and the driver functions for each device. Later chapters describe how device drivers are organized, and provide examples. The previous chapters illustrate how a file system fits into the device paradigm, and illustrate the concept of pseudo-devices.

This chapter considers a generalization of device names. It explains how names can be viewed syntactically, and shows that both devices and files can be represented in a single unified namespace.

21.2 Transparency And A Namespace Abstraction

Transparency forms one of the fundamental principles in operating system design:

> *Whenever possible, applications should remain unaware of implementation details such as the location of an object or its representation.*

For example, when an application creates a new process, the application does not need to know the location of the code or the location of the stack that is allocated. Similarly, when an application opens a local file, the application does not need to know which disk blocks the file occupies.

In terms of file access, the Xinu paradigm seems to violate the principle of transparency because it requires the user to specify a file system name when opening a file. For example, the master device for the local file system is named *LFILESYS*. When a Xinu system includes a remote file system, the violation of transparency becomes obvious: a programmer must also know the master device for the remote file system, *RFILESYS*, and must choose between local and remote files. Furthermore, a file name must be chosen to match the naming convention used on the specified file system.

How can we add transparency to file and device naming? The answer lies in a high-level abstraction called a *namespace*. Conceptually, a namespace provides a uniform set of names that knits together apparently diverse file naming schemes into a single unified whole, allowing users to open files and devices without knowing their location. The Unix operating system uses a *directory abstraction* to provide a namespace: local files, remote files, and devices are each assigned names in a hierarchical directory namespace. For example, the name */dev/console* usually corresponds to the system console device and the name */dev/usb* corresponds to the USB device.

Xinu takes a novel approach to the namespace abstraction by separating the namespace mechanism from the underlying files and directories. Furthermore, Xinu uses a *syntactic* approach, which means that the namespace examines names without understanding their meaning. What makes our implementation of the namespace especially fascinating is its combination of simplicity and power. By thinking of names as strings, we can understand their similarity. By using the relationship between prefix strings and trees, we can manipulate names easily. By following the principle of access transparency, we can improve the system dramatically. Adding just a small layer of software to existing mechanisms will allow us to unify naming completely.

Before looking at namespace mechanisms, we will review a few examples of file naming to understand the problem at hand. Following the discussion of file names, we will look at a general-purpose syntactic naming scheme, and then examine a simpler, less general solution. Finally, we will examine an implementation of the simplified scheme.

21.3 Myriad Naming Schemes

The problem designers face when inventing a namespace can be summarized: they must glue together myriad unrelated naming schemes, each of which has evolved into a self-contained system. On some systems, file names specify the storage device on which the file resides. On others, the filename includes a suffix that tells the type of the file (older systems used suffixes to specify a version for the file). Other systems map all files into a single flat namespace in which each name is merely a string of alphanumeric characters. The following sections give examples of file names on several

systems, and help the reader understand the types and formats of names our namespace must accommodate.

21.3.1 MS-DOS

Names in MS-DOS consist of two parts: a device specification and a file name. Syntactically, an MS-DOS name has the form *X*:*file*, where *X* is a single letter that designates the device on which the file resides and *file* is the name of the file. Typically, the letter *C* denotes the system hard disk, which means a name like *C:abc* refers to file *abc* on the hard disk.

21.3.2 Unix

Unix organizes files into a hierarchical, tree structured directory system. A file name is either relative to the current directory or a full *path name* that specifies a path from the root of the directory tree to a file.

Syntactically, full path names consist of slash-separated components, where each intermediate component specifies a directory and the final component specifies a file. Thus, the Unix name /*homes*/*xinu*/*x* refers to file *x* in subdirectory *xinu*, which is found in subdirectory *homes*, which is contained in the root directory. The root directory itself is named by a single slash (/). Notice that the prefix /*homes*/*xinu*/ refers to a directory, and that the names of all files in that directory share the prefix.

The importance of the *prefix property* will become apparent later. For now, it is sufficient to remember that the tree structure relates to a name prefix:

> *When components in a file name specify a path through a tree-structured directory, the names of all files that reside in the same directory share a common prefix that denotes the directory.*

21.3.3 V System

A research operating system known as the *V system* allowed a user to specify a *context* and a name; the system used the context to resolve the name. The syntax used brackets to enclose the context. Thus, [*ctx*] *abc* refers to file *abc* in context *ctx*. Usually, one thinks of each context as a set of files on a particular remote file server.

21.3.4 IBIS

The research operating system, *IBIS*, provides yet another syntax for multiple-machine connections that has been adopted for use with Linux commands, such as *scp*. In IBIS, names have the form *machine*:*path*, where *machine* denotes a particular computer system, and *path* is the file name on that machine (e.g., a Unix full path name).

21.4 Naming System Design Alternatives

We seek a single naming system that provides a unified view of all possible file names, independent of the location of the file or the operating system under which it resides. It seems that a designer could choose between two basic approaches in solving the problem: define yet another file naming scheme, or adopt an existing naming scheme. Surprisingly, the Xinu namespace uses neither of these two approaches. Instead, it adds a syntactic naming mechanism that accommodates many underlying naming schemes, while allowing the user to choose a uniform interface to the naming software. The namespace software maps names that the user supplies into names appropriate for the underlying system.

A naming mechanism that accommodates many underlying schemes has several advantages. First, it allows the designer to integrate existing file systems and devices into a single, uniform namespace, even when implemented by remote servers on a set of heterogeneous systems. Second, it permits designers to add new devices or file systems without requiring recompilation of the application programs that use them. Third, it avoids two unattractive extremes. At one extreme, choosing the simplest naming scheme ensures that all file systems can handle the names, but means that the user cannot take advantage of the complexity offered by some servers. At the other extreme, choosing a naming scheme that encompasses the most complex cases means that an application which takes advantage of the complexity may not be able to run on a less sophisticated file system.

21.5 Thinking About Names Syntactically

To understand how to handle names, think of them syntactically: a name is merely a string of characters. Consequentially, a namespace can be created that transforms strings. The namespace does not need to provide files and directories, nor does it need to understand the semantics of each underlying file system. Instead, the namespace maps strings from a uniform representation chosen by the user into strings appropriate for each particular subsystem. For example, the namespace might translate the string *clf* into the string *C*:*long_file_name*.

What makes a syntactic namespace powerful? Syntactic manipulation is both natural and flexible. Thus, it is easy to specify and understand as well as easy to adapt to many underlying naming schemes. A user can imagine a consistent set of names, and can use the namespace software to translate them into the forms required by underlying file systems. For example, suppose a system has access to a local file system that uses MS-DOS naming and a remote file system that uses Unix full path names. The user might adopt the Unix full path name syntax for all names, making the local disk names start with /*local*. In such a scheme, the name /*local*/*abc* would refer to file *abc* on the local hard drive, while the name /*etc*/*passwd* would refer to a remote file. The namespace must translate /*local*/*abc* into *C*:*abc* so the local MS-DOS file system can

understand it, but would pass */etc/passwd* on to the remote Unix file system without change.

21.6 Patterns And Replacements

Exactly how should a syntactic namespace operate? One convenient method uses a *pattern* string to specify the name syntax and a *replacement* string to specify the mapping. For example, consider the pattern replacement pair:

```
"/local"  "C:"
```

which means "translate all occurrences of the string */local* into the string *C:*".

How should patterns be formed? Patterns that consist of literal strings cannot specify replacement unambiguously. In the previous example, the pattern works well on strings like */local/x*, but it fails on strings like */homes/local/bin* because */local* is an internal substring that should not be changed. To be effective, more powerful patterns are needed. For example, Unix pattern matching tools introduce meta-characters that specify how matching should be performed. A *carat* (sometimes called *up-arrow*) matches the beginning of a string. Thus, the Unix pattern:

```
"^/local"  "C:"
```

specifies that */local* only matches at the beginning of a string. Unfortunately, implementations that allow arbitrary patterns and replacements tend to be cumbersome and the patterns become difficult to read. A more efficient solution is needed.

21.7 Prefix Patterns

The problem at hand is to find a useful pattern-replacement mechanism that allows the user to specify how names map onto a subsystem without introducing more complexity than is needed. Before thinking about complex patterns, consider what can be done with patterns that consist of literal strings. The key is to imagine files organized into a hierarchy, and to use the prefix property to understand why patterns should specify prefixes.

In a hierarchy, name prefixes group files into subdirectories, making it easy to define the relationship between names and the underlying file systems or devices. Furthermore, each prefix can be represented by a literal string. The point is:

> *Restricting name replacement to prefixes means it is possible to use literal strings to separate underlying file systems into distinct parts of a name hierarchy.*

21.8 Implementation Of A Namespace

A concrete example will clarify the details of how a syntactic namespace uses the pattern-replacement paradigm, and will show how the namespace hides subsystem details. In the example, patterns will consist of fixed strings, and only prefixes will be matched. Later sections discuss alternative implementations and generalizations.

The example implementation of a namespace consists of a pseudo-device, *NAMESPACE*, that programs use to open a named object. An application program invokes *open* on the *NAMESPACE* device, passing a name and mode as arguments. The *NAMESPACE* pseudo-device uses a set of prefix patterns to transform the name into a new name, and then passes the new name to the appropriate underlying device through a call to *open*. We will see that all files and devices can be part of the namespace, meaning that an application never needs to open a device other than the *NAMESPACE* pseudo-device.

The next sections present the namespace software, beginning with declarations of the basic data structures and culminating in the definition of the *NAMESPACE* pseudo-device. Following the declarations, two functions are presented that transform names according to the prefix patterns. The two functions are used by the most important piece of namespace software: a function that implements *open* for the *NAMESPACE* pseudo-device. Later sections show examples that illustrate how the *NAMESPACE* is used.

21.9 Namespace Data Structures And Constants

File *name.h* contains declarations for the data structures and constants used in the Xinu namespace.

```
/* name.h */

/* Constants that define the namespace mapping table sizes */

#define NM_PRELEN       64              /* Max size of a prefix string  */
#define NM_REPLLEN      96              /* Maximum size of a replacement*/
#define NM_MAXLEN       256             /* Maximum size of a file name  */
#define NNAMES          40              /* Number of prefix definitions */

/* Definition of the name prefix table that defines all name mappings */

struct  nmentry {                       /* Definition of prefix table   */
        char    nprefix[NM_PRELEN];     /* Null-terminated prefix       */
        char    nreplace[NM_REPLLEN];   /* Null-terminated replacement  */
        did32   ndevice;                /* Device descriptor for prefix */
};
```

```
extern  struct  nmentry nametab[];      /* Table of name mappings       */
extern  int32   nnames;                 /* Number of entries allocated  */
```

The principle data structure is array *nametab*, which holds up to *NNAMES* entries. Each entry consists of a prefix pattern string, a replacement string, and a device ID. External integer *nnames* holds a count of the valid entries in *nametab*.

21.10 Adding Mappings To The Namespace Prefix Table

Function *mount* is used to add mappings to the prefix table. As expected, *mount* takes three arguments: a prefix string, a replacement string, and a device ID. File *mount.c* contains the code.

```
/* mount.c - mount, namlen */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  mount  -  Add a prefix mapping to the name space
 *------------------------------------------------------------------------
 */
syscall mount(
          char          *prefix,        /* Prefix to add                */
          char          *replace,       /* Replacement string           */
          did32         device          /* Device ID to use             */
)
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  nmentry *namptr;        /* Pointer to unused table entry*/
        int32   psiz, rsiz;             /* Sizes of prefix & replacement*/
        int32   i;                      /* Counter for copy loop        */

        mask = disable();

        psiz = namlen(prefix, NM_PRELEN);
        rsiz = namlen(replace, NM_REPLLEN);

        /* If arguments are invalid or table is full, return error */

        if ( (psiz == SYSERR)   || (rsiz == SYSERR) ||
             (isbaddev(device)) || (nnames >= NNAMES) ) {
                restore(mask);
                return SYSERR;
        }
```

```
        /* Allocate a slot in the table */

        namptr = &nametab[nnames];      /* Next unused entry in table   */

        /* copy prefix and replacement strings and record device ID */

        for (i=0; i<psiz; i++) {        /* Copy prefix into table entry */
                namptr->nprefix[i] = *prefix++;
        }

        for (i=0; i<rsiz; i++) {        /* Copy replacement into entry  */
                namptr->nreplace[i] = *replace++;
        }

        namptr->ndevice = device;       /* Record the device ID         */

        nnames++;                       /* Increment number of names    */

        restore(mask);
        return OK;
}


/*------------------------------------------------------------------------
 *  namlen  -  Compute the length of a string stopping at maxlen
 *------------------------------------------------------------------------
 */
int32   namlen(
          char          *name,          /* Name to use                  */
          int32         maxlen          /* Maximum length (including a  */
                                        /*   NULLCH)                    */
)
{
        int32   i;                      /* Count of characters found    */

        /* Search until a null terminator or length reaches max */

        for (i=0; i < maxlen; i++) {
                if (*name++ == NULLCH) {
                        return i+1;     /* Include NULLCH in length */
                }
        }
        return SYSERR;
}
```

If any of the arguments are invalid or the table is full, *mount* returns *SYSERR*. Otherwise, it increments *nnames* to allocate a new entry in the table and fills in the values.

21.11 Mapping Names With The Prefix Table

Once a prefix table has been created, name translation can be performed. Mapping consists of finding a prefix match and substituting the corresponding replacement string. Function *nammap* performs translation. The code can be found in file *nammap.c*:

```
/* nammap.c - nammap, namrepl, namcpy */

#include <xinu.h>

status  namcpy(char *, char *, int32);
did32   namrepl(char *, char[]);

/*------------------------------------------------------------------------
 *  nammap  -  Using namespace, map name to new name and new device
 *------------------------------------------------------------------------
 */
devcall nammap(
         char   *name,                  /* The name to map              */
         char   newname[NM_MAXLEN],     /* Buffer for mapped name       */
         did32 namdev                   /* ID of the namespace device   */
        )
{
        did32   newdev;                 /* Device descriptor to return  */
        char    tmpname[NM_MAXLEN];     /* Temporary buffer for name    */
        int32   iter;                   /* Number of iterations         */

        /* Place original name in temporary buffer and null terminate */

        if (namcpy(tmpname, name, NM_MAXLEN) == SYSERR) {
                return SYSERR;
        }

        /* Repeatedly substitute the name prefix until a non-namespace  */
        /*   device is reached or an iteration limit is exceeded        */

        for (iter=0; iter<nnames ; iter++) {
                newdev = namrepl(tmpname, newname);
                if (newdev != namdev) {
```

```
                        namcpy(tmpname, newname, NM_MAXLEN);
                        return newdev;  /* Either valid ID or SYSERR    */
                }
        }
        return SYSERR;
}

/*------------------------------------------------------------------------
 *  namrepl  -  Use the name table to perform prefix substitution
 *------------------------------------------------------------------------
 */
did32   namrepl(
         char   *name,                  /* Original name                */
         char   newname[NM_MAXLEN]      /* Buffer for mapped name       */
        )
{
        int32   i;                      /* Iterate through name table   */
        char    *pptr;                  /* Walks through a prefix       */
        char    *rptr;                  /* Walks through a replacement  */
        char    *optr;                  /* Walks through original name  */
        char    *nptr;                  /* Walks through new name       */
        char    olen;                   /* Length of original name      */
                                        /*   including the NULL byte    */
        int32   plen;                   /* Length of a prefix string    */
                                        /*   *not* including NULL byte  */
        int32   rlen;                   /* Length of replacment string  */
        int32   remain;                 /* Bytes in name beyond prefix  */
        struct  nmentry *namptr;        /* Pointer to a table entry     */

        /* Search name table for first prefix that matches */

        for (i=0; i<nnames; i++) {
                namptr = &nametab[i];
                optr = name;            /* Start at beginning of name   */
                pptr = namptr->nprefix; /* Start at beginning of prefix */

                /* Compare prefix to string and count prefix size */

                for (plen=0; *pptr != NULLCH ; plen++) {
                        if (*pptr != *optr) {
                                break;
                        }
                        pptr++;
                        optr++;
                }
```

```
                if (*pptr != NULLCH) {  /* Prefix does not match */
                        continue;
                }

                /* Found a match - check that replacement string plus   */
                /* bytes remaining at the end of the original name will */
                /* fit into new name buffer.  Ignore null on replacement*/
                /* string, but keep null on remainder of name.          */

                olen = namlen(name ,NM_MAXLEN);
                rlen = namlen(namptr->nreplace,NM_MAXLEN) - 1;
                remain = olen - plen;
                if ( (rlen + remain) > NM_MAXLEN) {
                        return (did32)SYSERR;
                }

                /* Place replacement string followed by remainder of    */
                /*   original name (and null) into the new name buffer  */


                nptr = newname;
                rptr = namptr->nreplace;
                for (; rlen>0 ; rlen--) {
                        *nptr++ = *rptr++;
                }
                for (; remain>0 ; remain--) {
                        *nptr++ = *optr++;
                }
                return namptr->ndevice;
        }
        return (did32)SYSERR;
}

/*------------------------------------------------------------------------
 *  namcpy  -  Copy a name from one buffer to another, checking length
 *------------------------------------------------------------------------
 */
status  namcpy(
         char           *newname,       /* Buffer to hold copy          */
         char           *oldname,       /* Buffer containing name       */
         int32          buflen          /* Size of buffer for copy      */
        )
{
        char    *nptr;                  /* Point to new name            */
        char    *optr;                  /* Point to old name            */
```

```
        int32   cnt;                    /* Count of characters copied   */

        nptr = newname;
        optr = oldname;

        for (cnt=0; cnt<buflen; cnt++) {
                if ( (*nptr++ = *optr++) == NULLCH) {
                        return OK;
                }
        }
        return SYSERR;          /* Buffer filled before copy completed  */
}
```

The most interesting aspect of *nammap* arises because it allows multiple mappings. In particular, because the namespace is a pseudo-device, it is possible for a user to specify a mapping back onto the *NAMESPACE* device. For example, consider the following two entries in *nametab*:

```
"/local/"   ""          LFILESYS
"LFS:"      "/local/"   NAMESPACE
```

The first entry specifies that if a name begins with */local/*, the prefix is removed and the name is passed to the local file system. The second entry specifies that *LFS:* is an abbreviation for */local/*. That is, the prefix *LFS:* is replaced by */local/* and the resulting string is passed back to the NAMESPACE device for another round of mapping.

Of course, recursive mapping can be dangerous. Consider what can happen if a user adds the following to the namespace:

```
"/x"        "/x"        NAMESPACE
```

When presented with a name */xyz*, a naive implementation will find prefix */x*, make the substitution, and call *open* on the NAMESPACE device, causing an infinite recursion. To avoid the problem, our implementation iterates through NAMESPACE replacements and limits the total iterations. In particular, the code only permits one iteration for each prefix in *nametab* (i.e., each prefix can be substituted at most once). Of course, *nammap* also limits the size of a name: if a replacement would expand the name to more than *NM_MAXLEN* characters, *nammap* stops and returns *SYSERR*.

Nammap begins by copying the original name into local array *tmpname*. It then iterates until the name has been mapped to a device other than the NAMESPACE or the iteration limit is reached. During each iteration, *nammap* calls function *namrepl* to look up the current name and form a replacement.

Function *namrepl* implements a basic *replacement policy*. Our example replacement policy is simplistic: *namrepl* searches the table linearly. A search always begins with the first entry in the table, and stops as soon as a prefix in the table matches the

string supplied by argument *name*. Once searching has stopped, *nammap* forms a mapped name in argument *newname* by appending the unmatched portion of the original name onto the replacement string. It then returns the device ID from the table entry. A later section explains that the design has consequences for users.

21.12 Opening A Named File

Once *nammap* is available, constructing the upper-half *open* routine for the namespace pseudo-device becomes trivial. Recall that the basic goal is to define a namespace pseudo-device, *NAMESPACE*, such that opening the device causes the system to open the appropriate underlying device. Once a name has been mapped and a new device identified, *namopen* merely invokes *open*. The code is contained in file *namopen.c*.

```
/* namopen.c - namopen */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  namopen  -  Open a file or device based on the name
 *------------------------------------------------------------------------
 */
devcall namopen(
          struct dentry *devptr,        /* Entry in device switch table */
          char  *name,                  /* Name to open                 */
          char  *mode                   /* Mode argument                */
        )
{
        char    newname[NM_MAXLEN];     /* Name with prefix replaced    */
        did32   newdev;                 /* Device ID after mapping      */

        /* Use namespace to map name to a new name and new descriptor */

        newdev = nammap(name, newname, devptr->dvnum);

        if (newdev == SYSERR) {
                return SYSERR;
        }

        /* Open underlying device and return status */

        return  open(newdev, newname, mode);
}
```

21.13 Namespace Initialization

How should initial values be assigned to the prefix table? There are two possible approaches: an initialization function could assign initial mappings when it creates the namespace data structure, or the initialization function could leave the table empty and require the user to add mappings. We chose the former approach.

In terms of initialization, the mechanism should be clear. Because the namespace has been designed as a pseudo-device, the files resemble a device driver. In particular, the namespace device includes an initialization function that the system calls when devices are initialized (i.e., when the system calls *init* for the device at system startup).

Deciding how to initialize a prefix table can be difficult. Therefore, we will examine the initialization function to see how it constructs a prefix table, and defer the discussion of actual prefixes until later sections. File *naminit.c* contains the code for the *naminit* function:

```
/* naminit.c - naminit */

#include <xinu.h>

#ifndef RFILESYS
#define RFILESYS        SYSERR
#endif

#ifndef FILESYS
#define FILESYS         SYSERR
#endif

#ifndef LFILESYS
#define LFILESYS        SYSERR
#endif

struct  nmentry nametab[NNAMES];        /* Table of name mappings       */
int32   nnames;                         /* Number of entries allocated  */

/*------------------------------------------------------------------------
 *  naminit  -  Initialize the syntactic namespace
 *------------------------------------------------------------------------
 */
status  naminit(void)
{
        did32   i;                      /* Index into devtab            */
        struct  dentry *devptr;         /* Pointer to device table entry*/
        char    tmpstr[NM_MAXLEN];      /* String to hold a name        */
        status  retval;                 /* Return value                 */
        char    *tptr;                  /* Pointer into tempstring      */
        char    *nptr;                  /* Pointer to device name       */
```

```
char    devprefix[] = "/dev/";  /* Prefix to use for devices    */
int32   len;                    /* Length of created name       */
char    ch;                     /* Storage for a character      */

/* Set prefix table to empty */

nnames = 0;

for (i=0; i<NDEVS ; i++) {
        tptr = tmpstr;
        nptr = devprefix;

        /* Copy prefix into tmpstr */

        len = 0;
        while ((*tptr++ = *nptr++) != NULLCH) {
                len++;
        }
        tptr--; /* Move pointer to position before NULLCH */
        devptr = &devtab[i];
        nptr = devptr->dvname;  /* Move to device name */

        /* Map device name to lower case and append */

        while(++len < NM_MAXLEN) {
                ch = *nptr++;
                if ( (ch >= 'A') && (ch <= 'Z')) {
                        ch += 'a' - 'A';
                }
                if ( (*tptr++ = ch) == NULLCH) {
                        break;
                }
        }

        if (len > NM_MAXLEN) {
                kprintf("namespace: device name %s too long\r\n",
                                devptr->dvname);
                continue;
        }

        retval = mount(tmpstr, NULLSTR, devptr->dvnum);
        if (retval == SYSERR) {
                kprintf("namespace: cannot mount device %d\r\n",
                                devptr->dvname);
                continue;
        }
}
```

```
        /* Add other prefixes (longest prefix first) */

        mount("/dev/null",      "",         NULLDEV);
        mount("/remote/",       "remote:",  RFILESYS);
        mount("/local/",        NULLSTR,    LFILESYS);
        mount("/dev/",          NULLSTR,    SYSERR);
        mount("~/",             NULLSTR,    LFILESYS);
        mount("/",              "root:",    RFILESYS);
        mount("",               "",         LFILESYS);

        return OK;
}
```

Ignore the specific prefix and replacement names and look only at how straightforward initialization is. After setting the number of valid entries to zero, *naminit* calls *mount* to add entries to the prefix table, where each entry contains a prefix pattern, replacement string, and device id. The for loop iterates through the device switch table. For each device, it creates a name of the form */dev/xxx*, where *xxx* is the name of the device mapped into lower case. Thus, it creates an entry for */dev/console* that maps to the CONSOLE device. If a process calls:

```
d = open(NAMESPACE,"/dev/console","rw");
```

the namespace will invoke *open* on the CONSOLE device and return the result.

21.14 Ordering Entries In The Prefix Table

The Xinu name replacement policy affects users. To understand how, recall that *namrepl* uses sequential lookup. Therefore, a user must mount names so that sequential lookup produces the expected outcome. In particular, our implementation does not prohibit overlapping prefixes, and does not warn users if overlaps occur. Consequently, if overlapping prefixes occur, a user must ensure that the longest prefix appears earlier in the table than shorter prefixes. As an example, consider what happens if the table contains two entries as Figure 21.1 illustrates.

Prefix	Replacement	Device
"x"	"" (null string)	LFILESYS
"xyz"	"" (null string)	RFILESYS

Figure 21.1 Two entries in a prefix table; the order must be swapped or the second entry will never be used.

The first entry maps prefix *x* to a local file system, and the second entry maps prefix *xyz* to a remote file system. Unfortunately, because *namrepl* searches the table sequentially, any file name that starts with *x* will match the first entry and will be mapped to the local file system. The second entry will never be used. If the two are reversed, however, file names starting with *xyz* will map onto the remote file system, and other names starting with *x* will map to the local file system. The point is:

Because our implementation searches the table of prefixes sequentially and does not detect overlapping prefixes, a user must insert prefixes in reverse order by length, ensuring that the system will match the longest prefix first.

21.15 Choosing A Logical Namespace

It is tempting to think of a namespace as merely a mechanism that can be used to abbreviate long names. However, focusing on the mechanism can be misleading. The key to choosing meaningful prefix names lies in thinking of a hierarchy into which files can be placed. Then, the namespace design defines the organization of the hierarchy.

Rather than thinking of the namespace as a mechanism that abbreviates names, we think of all names being organized into a hierarchy. Entries in the namespace are chosen to implement the desired hierarchy.

Imagine, for a minute, a system that can access files on a local disk as well as files on a remote server. Do not think about how to abbreviate specific file names; think instead of how to organize the files. Three possible organizations come to mind as Figure 21.2 shows.

Figure 21.2 Three possible hierarchical organizations of local and remote files: (a) local and remote files at the same level, (b) remote files in a subdirectory of local files, and (c) local files as a subdirectory of remote files.

As in the figure, local and remote files could be placed at equal, but distinct positions in the hierarchy. Alternatively, the local system could form the main part of the hierarchy with remote files as a sub-hierarchy, or the remote files could form the main hierarchy with local files as a sub-hierarchy. Among the choices, the size of the two file systems and the frequency of access may help determine which organization is preferable. For example, if the remote file system has thousands of files while the local file system has only ten, it may be natural to think of the remote file system as the main hierarchy with the local files grafted onto a sub-hierarchy.

21.16 A Default Hierarchy And The Null Prefix

The Xinu namespace software design can easily support any of the hierarchies shown in Figure 21.2. In particular, *mount* permits the user to choose one subsystem as the default, and organize the remaining files with respect to that hierarchy.

How does a subsystem become the default? First, the prefix for the subsystem must be such that it matches all names not matched by other table entries. The null prefix provides guaranteed matching for our example namespace. Second, the default entry, which carries the null prefix, must be examined only after all other prefixes have been tested. Because *nammap* searches the prefix table sequentially, the default must be placed at the end of the table. If any other entry matches, *namrepl* will follow the match.

Look at *naminit* again to see how the local file system becomes the default. The final call to *mount* inserts the default mapping with a null prefix. Thus, any name that does not match another prefix will refer to the local file system.

21.17 Additional Object Manipulation Functions

Although it appears to organize all names into a single, unified hierarchy, the namespace illustrated above does not provide all needed functionality. To understand why, observe that the code only handles the case of opening a named object. Other operations on named objects are also possible:

- Testing the existence of an object
- Changing the name of an object
- Deleting an object

Testing the existence of an object. Often software needs to test the existence of an object without otherwise affecting the object. It may seem that the following test could be used to determine whether an object exists.

```
dev = open(NAMESPACE", "object", "r");
if (dev == SYSERR) {
        ...object does not exist
} else {
        close(dev);
        ...object exists
}
```

Unfortunately, a call to *open* can have side effects. For example, opening a network interface device can cause the system to declare the interface to be available for packet transfer. Thus, opening and then closing the device may cause packet transfer to begin, even if a process has specifically disabled the interface. To avoid side effects, additional functionality is needed.

Changing the name of an object. Most file systems allow a user to rename files. However, two problems arise when the namespace is used. First, because users view all files through the namespace, requests to rename files can be ambiguous: should the name of the underlying file be changed, or should the mapping in the namespace be changed? Although it is possible to include an escape mechanism that allows a user to distinguish between abstract names and names used in the underlying system, doing so is dangerous because changes to underlying names may no longer map through the namespace. Second, if a user specifies changing name α to name β, it may turn out that string β maps to a local file system and α maps to a remote file system. Thus, even though a user sees a unified hierarchy, a renaming operation may not be allowed (or may involve a file copy).

Deleting an object. The reasoning given above applies to object deletion as well. That is, because a user views all names through the namespace, a request to delete an object must map the name through the namespace to determine the underlying file system.

How should deletion, renaming, and testing the existence of an object be implemented? Three possibilities arise: separate functions could be created for each operation, the device switch table could be expanded, or the *control* functions could be augmented with additional operations. To understand the first approach, separate functions, imagine a function *delete_obj* that takes a name as an argument. The function would use the namespace to map the name to an underlying device, and then invoke the appropriate deletion operation on the device. The second approach consists of expanding the device switch table to add additional high-level functions, such as *delete*, *rename*, and an *existence test*. That is, in addition to *open*, *read*, *write*, and *close* operations, add new operations that implement the additional functionality. The third approach consists of adding functionality to the *control* function. For example, we could specify that if a subsystem implements object deletion, the driver function that implements *control* must honor a *DELETE* request. Xinu uses a mixture of the first and third approaches. Exercises ask the reader to consider the advantages and disadvantages of expanding the device switch table.

21.18 Advantages And Limits Of The Namespace Approach

A syntactic namespace isolates programs from the underlying devices and file systems, allowing a naming hierarchy to be imagined or changed without changing the underlying systems. To appreciate the power of a namespace, consider a system that keeps temporary files on a local disk and uses the prefix */tmp/* to distinguish them from other files. Moving temporary files to the remote file system consists of changing the namespace entry that specifies how to handle prefix */tmp/*. Because programs always use the namespace when they refer to files, all programs continue to operate correctly with no change to the source code. The key point is:

> *A namespace permits the conceptual naming hierarchy to be reorganized without requiring recompilation of programs that use it.*

Namespace software that uses only prefix patterns cannot handle all hierarchical organizations or file mappings. For example, in some Unix systems, the name */dev/tty* refers to a process's control terminal, which a server should not use. The namespace can prevent accidental access by mapping the prefix */dev/tty* onto device ID *SYSERR*. Unfortunately, such a mapping prevents the client from accessing other entries that share the same prefix (e.g., */dev/tty1*).

Using fixed strings as prefix patterns also prevents the namespace from changing separator characters when they occur in the middle of names. For example, suppose a computer has two underlying file systems, one of which follows the Unix convention of using the slash character to separate components along a path, while the other uses the backslash character to separate components. Because it deals only with prefix patterns, our example namespace cannot map slashes to backslashes or vice versa unless all possible prefixes are stored in the namespace.

21.19 Generalized Patterns

Many of the namespace limitations can be overcome by using more general patterns than were described near the beginning of the chapter. For example, if it is possible to specify a *full string match* instead of just a prefix match, the problem of distinguishing a name like */dev/tty* from the name */dev/tty1* can be solved. Full match and prefix match can be combined: *mount* can be modified to have an additional argument that specifies the type of match, and the value can be stored in the table entry.

Generalizing patterns to allow more than fixed strings solves additional problems and keeps all matching information in the pattern itself. For example, suppose characters have special meanings in a pattern as defined in Figure 21.3:†

†The pattern matching given here corresponds to that used in the Unix *sed* command.

Character	Meaning
↑	match beginning of string
$	match end of string
.	match any single character
*	repeat 0 or more of a pattern
\	take next character in pattern literally
other	self match as in a fixed string

Figure 21.3 An example definition of generalized patterns.

Thus, a pattern like ↑/*dev*/*tty*$ specifies a full match of the string /*dev*/*tty*, while a pattern like \$ matches a dollar sign that may be embedded in the string.

Two additional rules are necessary to make generalized pattern matching useful in the namespace. First, we assume the left-most possible match will be used. Second, we assume that among all left-most matches, the longest will be selected. The exercises suggest how to use these generalized patterns to map the names that fixed prefixes cannot handle.

21.20 Perspective

The syntax of names has been studied and debated extensively. At one time, each operating system had its own naming scheme, and a variety of naming schemes flourished. However, once hierarchical directory systems became prevalent, most operating systems adopted a hierarchical naming scheme, and the only differences arise over small details, such as whether a forward slash or backslash is used to separate components.

As our design points out, naming is conceptually separate from the underlying file and I/O systems, and allows a designer to impose a consistent namespace across all underlying facilities. However, the use of a syntactic approach has disadvantages as well as advantages. The chief problems arise from semantics: although it provides the appearance of uniformity, the namespace introduces ambiguity and confuses semantics. For example, if an object is renamed, should the namespace be modified or should the name of the underlying object be changed? If the namespace maps two prefixes to the same underlying file system, applications that use separate prefixes could inadvertently access the same object. If two names map to different underlying file systems, an operation such as *move* that references the names may not be possible or may not work as expected. Even an operation such as *delete* may have unexpected semantics (e.g., deleting a local object may move it to the trash, while deleting a remote object permanently removes the object).

21.21 Summary

Dealing with file names is difficult, especially if the operating system supports multiple underlying naming schemes. One way to solve the naming problem employs a layer of namespace software between applications and the underlying file systems. The namespace does not implement files itself, but merely treats names as strings, mapping them into forms appropriate for underlying systems based on information in a mapping table.

We examined the implementation of a syntactic namespace that uses a pattern-replacement scheme in which patterns are fixed strings representing name prefixes. The software includes a *mount* that installs a mapping, as well as functions like *nammap* that map names into their target form. Our example namespace uses a *NAMESPACE* pseudo-device that users specify when opening a file. The *NAMESPACE* pseudo-device maps the specified file name, and then opens the designated file.

The namespace software is both elegant and powerful. With only a few functions and the simplistic notion of prefix matching, it can accommodate many naming schemes. In particular, it accommodates a remote file system, a local file system, and a set of devices. However, our simplistic version cannot handle all possible mappings. To provide for more complex naming systems, the notion of pattern must be generalized. One possible generalization assigns special meanings to some characters in the pattern.

EXERCISES

21.1 Should users have access to both *nammap* and *namrepl*? Why or why not?

21.2 Look carefully at the code in *mount*. Will the prefix and replacement strings in *nametab* always include a NULL byte? Why or why not?

21.3 Is it possible to modify *mount* so it refuses to mount prefix-replacement pairs that can potentially cause an infinite loop? Why or why not?

21.4 What is the minimum number of prefix-replacement pairs that can cause *nammap* to exceed the maximum string length?

21.5 Minimize the code in *namopen* by replacing the body with a single statement consisting of two function calls.

21.6 Implement an upper-half *control* function for the *NAMESPACE* pseudo-device, and make *nammap* a control function.

21.7 Implement generalized pattern matching. Refer to the Unix *sed* command for additional ways to define pattern match characters.

21.8 Build a namespace that has both prefix matches and full string matches.

21.9 Suppose a namespace uses fixed string patterns, but allows full string matching in addition to the current prefix matching. Are there instances when it makes sense to have a full string pattern identical to a prefix pattern? Explain.

21.10 What additional file manipulation primitives might be needed beside *rename*, *delete*, and *existence test*?

21.11 Implement a function *unmount* that removes a prefix from the mapping table.

21.12 As written, the namespace functions rely on a user to install patterns in longest-prefix order. Rewrite the code to allow a user to mount patterns in an arbitrary order, and arrange the table by prefix length.

21.13 Rewrite function *mount* (either the original or the version in the previous exercise) to check for overlapping patterns, and report an error.

21.14 Examine the *Network File System* technology used with Linux systems that allows remote files to be mapped into the directory hierarchy. What basic file operation is not permitted across file systems?

Chapter Contents

22.1 Introduction, 573
22.2 Bootstrap: Starting From Scratch, 573
22.3 An Example Of Booting Over A Network, 574
22.4 Operating System Initialization, 575
22.5 Xinu Initialization, 576
22.6 Xinu System Startup, 579
22.7 Transforming A Program Into A Process, 583
22.8 Perspective, 584
22.9 Summary, 584

22

System Initialization

Only by avoiding the beginning of things can we escape their end.

— Cyril Connolly

22.1 Introduction

Initialization is the last step of the design process. Designers create a system by thinking about the system in an executing state and postponing the details of how to get the system started. Thinking about initialization early has the same bad effect as worrying about optimization early: it tends to impose unnecessary constraints on the design, and diverts the designer's attention from important issues to trivial ones.

This chapter describes the steps required to initialize the system, and explains the conceptual leap that initialization code makes in switching from a program that executes sequentially to a system that supports concurrent processes. We will see that no special hardware steps are involved, and understand that concurrency is an abstraction created entirely by operating system software.

22.2 Bootstrap: Starting From Scratch

Our discussion of initialization begins with a consideration of system termination. Everyone who has worked with a computer system knows that errant programs or malfunctions in the hardware lead to catastrophic failures popularly called *crashes*. A crash occurs when the hardware attempts an invalid operation because code or data for a given operation is incorrect. Users know that when a crash occurs, the contents of memory are lost and the operating system must be restarted, which often takes considerable time.

How can a computer that is devoid of operating system code spring into action and begin executing? It cannot. Somehow a program must be available before a computer can start. On the oldest machines, restarting was a painful process because a human operator entered the initial program through switches on the front panel. Switches were replaced by keyboards, then by I/O devices such as tapes and disks, and eventually by Read-Only Memory (ROM), and Flash memory.

Some embedded devices do more than store an initial program in Flash; they store the entire operating system, which means that the device can start executing immediately after receiving power (e.g., after the batteries are changed or the device is powered on). However, most computers take multiple steps when restarting. When power is applied, the hardware executes an initial startup program from Flash. Although it may include mechanisms that allow engineers to debug the hardware, an initial program is usually quite small — its primary function consists of loading and running a larger program. In a typical personal computer, for example, the startup program powers on devices (e.g., a display, keyboard, and disk), searches the disk for a bootable operating system image, copies the operating system image from disk into memory, and then jumps to the entry point of the operating system.

Computer systems that do not have permanent storage or embedded systems may use a network: the initial startup program initializes the network interface, and then uses the network to download the operating system image from a remote server. Some Ethernet hardware includes firmware and a small onboard processor that can download an image network over a network, store the image in memory, and then start the processor running the image.

In some cases, multiple steps are used. The initial startup program loads a larger, more powerful program, which then loads an operating system. Startup programs arranged in a sequence to load ever larger startup programs are often called *bootstraps*, and the entire process is known as *booting* the system.† Older names for the bootstrap process include *Initial Program Load* (*IPL*) and *cold start*.

22.3 An Example Of Booting Over A Network

The Galileo platform provides a specific example of a multi-stage bootstrap. Our lab at Purdue is designed so systems boot over a network. Unfortunately, a Galileo board does not have the hardware and software needed to boot over a network. Instead, when it receives power, a Galileo board runs an initial bootstrap program from the onboard Flash memory. The initial bootstrap, supplied by the vendor, can search local devices, find a specified image, download the image into memory, and start the processor executing the image. In particular, the initial bootstrap program can be configured to search the micro SD card for an image.

To enable network booting, we created a second bootstrap program that includes a network stack plus code to use the network to download an image. We placed a copy of the program on the SD card in each Galileo board. To boot a Galileo board, we power-cycle it. We built a special hardware system to handle the task: the system re-

†The terminology derives from the phrase "pulling one's self up by one's bootstraps," a seemingly impossible task.

moves power, waits a short time, and then restores power. Once a Galileo board receives power, the firmware begins the bootstrap process. The initial bootstrap program runs the second stage bootstrap, which uses the lab network to communicate with a server.

The server in our lab stores a set of Xinu images, one for each Galileo board. When our network bootstrap program sends a request to the server, the server uses the MAC address in the packet to choose the correct image, and then downloads the image over the network. Figure 22.1 lists the steps.

- Galileo board receives power and runs an initial bootstrap program from onboard Flash
- Initial bootstrap copies a second bootstrap from the SD card into memory
- Second bootstrap runs and uses the network to download a Xinu image
- Processor jumps to the downloaded Xinu image

Figure 22.1 The multi-step bootstrap used on the Galileo board to boot over a network.

We took an interesting approach to create a bootstrap program that downloads an image from a remote server. Instead of building code from scratch, we took a Xinu system, removed all the unneeded modules, and wrote a main program that used the *Trivial File Transfer Protocol* (*TFTP*) to download an image from a remote server. We call the program *Xboot* (for *Xinu bootstrap*).

If the Xboot code has been placed in memory, how can Xboot download another image without causing problems? The answer lies in a large memory. Instead of building Xboot to occupy the same memory locations as a standard Xinu image, Xboot is configured to run in high memory locations that lie beyond those used by a standard image. Thus, there is no conflict because Xboot can load a standard Xinu image into low memory without overwriting itself.

22.4 Operating System Initialization

The work of initialization does not end when the processor begins to execute the operating system image. Before it is ready to run processes, an operating system must initialize modules in the operating system as well as hardware devices. Figure 22.2 lists the initialization tasks.

- Perform initialization required by the hardware platform
- Initialize the memory management hardware and the free memory list
- Initialize each operating system module
- Load (if not present) and initialize a device driver for each device
- Start (or reset) each I/O device
- Transform from a sequential program to a concurrent system
- Create a null process
- Create a process to execute user code (e.g., a desktop)

Figure 22.2 Initialization tasks an operating system performs.

The most important step occurs after the hardware and operating system modules have been initialized: the operating system must undergo a metamorphosis, changing itself from a program that executes sequentially into an operating system that runs processes and supports concurrent execution. In the next sections, we will see what happens when Xinu boots, and see exactly how the transformation occurs.

22.5 Xinu Initialization

The underlying hardware handles some of the basic initialization tasks, and the bootstrap programs handle others. For example, if the firmware has initialized the bus and console I/O device, it may be possible for Xinu to use polled I/O (i.e., *kputc* and *kprintf*) immediately after execution begins. On the Galileo, where the address space is not contiguous, the startup program handles the task of finding available addresses. The startup program builds a list of available memory blocks and places the list in memory, making it available to Xinu.

Although some low-level initialization is completed before Xinu boots, an assembly language initialization function is still required. For example, Xinu must establish the runtime environment suitable for C. In our code, execution begins at label *start*, found in file *start.S*. The BeagleBone Black startup code provides an example:

```
/* start.S - start, bzero */

#include <armv7a.h>

/*------------------------------------------------------------------------
 * start  -  Initial entry point for a Xinu image (ARM)
 *------------------------------------------------------------------------
```

```
 */
        .text
        .globl  start                   /* Declare entry point global   */

start:
        /* Load the stack pointer with end of memory */

        ldr     sp, =MAXADDR

        /* Enable the Instruction Cache */

        mrc     p15, 0, r0, c1, c0, 0
        orr     r0, r0, #ARMV7A_C1CTL_I
        mcr     p15, 0, r0, c1, c0, 0

        /* Use bzero (below) to zero out the BSS area */

        ldr     r0, =edata
        ldr     r1, =end
        bl      bzero

        /* Call nulluser to initialize the Xinu system  */
        /*    (Note: the call never returns)            */

        bl      nulluser

        /* Function to zero memory (r0 is lowest addr; r1 is highest)   */

bzero:
        mov     r2, #0                  /* Round address to multiple    */
        add     r0, r0, #3              /*   of four by adding 3 and    */
        and     r0, r0, #0xFFFFFFFC     /*   taking the result module 4 */
bloop:  cmp     r0, r1                  /* Loop until last address      */
        bhs     bexit                   /*   has been reached           */
        str     r2, [r0]                /* Zero four-byte word of memory*/
        add     r0, r0, #4              /* Move to next word            */
        b       bloop                   /* Continue to iterate          */
bexit:  mov     pc, lr                  /* Return to caller             */
```

The code sets an initial stack pointer, enables the instruction cache, and calls *nulluser*. Setting the stack pointer is trivial because memory on the BeagleBone Black is contiguous and the highest address is specified by the hardware vendor. The code uses constant *MAXADDR*.

The code to enable the instruction cache seems cryptic, but the explanation is straightforward. On an ARM version 7a processor, the co-processor controls the cache. A bit in the co-processor control register determines whether the cache is enabled or disabled. The code fetches a copy of the control register, turns on the cache enable bit (constant *ARMv7A_C1CTL_I*), and stores the result back in the control register. File *armv7a.h*, which is included at the beginning of *start.S*, contains the definition of individual co-processor registers:

```
/* armv7a.h */

/* CPSR bits */

#define ARMV7A_CPSR_A   0x00000100      /* Imprecise data abort disable */
#define ARMV7A_CPSR_I   0x00000080      /* IRQ interrupts disable       */
#define ARMV7A_CPSR_F   0x00000040      /* FIQ interrupts disable       */
#define ARMV7A_CPSR_MM  0x0000001F      /* Processor Mode Mask          */
#define ARMV7A_CPSR_USR 0x00000010      /* Processor Mode = User        */
#define ARMV7A_CPSR_FIQ 0x00000011      /* Processor Mode = FIQ         */
#define ARMV7A_CPSR_IRQ 0x00000012      /* Processor Mode = IRQ         */
#define ARMV7A_CPSR_SPR 0x00000013      /* Processor Mode = Supervisor  */
#define ARMV7A_CPSR_ABT 0x00000017      /* Processor Mode = Abort       */
#define ARMV7A_CPSR_UND 0x0000001B      /* Processor Mode = Undefined   */
#define ARMV7A_CPSR_SYS 0x0000001F      /* Processor Mode = System      */
#define ARMv7A_CPSR_SCM 0x00000016      /* Processor Mode Secure Monitor*/

/* Coprocessor c1 - Control Register bits */

#define ARMV7A_C1CTL_V  0x00002000      /* Exception base addr control  */
#define ARMV7A_C1CTL_I  0x00001000      /* Instruction Cache enable     */
#define ARMV7A_C1CTL_C  0x00000004      /* Data Cache enable            */
#define ARMV7A_C1CTL_A  0x00000002      /* Strict alignment enable      */
#define ARMV7A_C1CTL_M  0x00000001      /* MMU enable                   */

/* Exception Vector Addresses */

#define ARMV7A_EV_START 0x4030CE00      /* Exception vector start addr  */
#define ARMV7A_EV_END   0x4030CE20      /* Exception vector end addr    */
#define ARMV7A_EH_START 0x4030CE24      /* Exception handler start addr */
#define ARMV7A_EH_END   0x4030CE40      /* Exception handler end addr   */
#define ARMV7A_IRQH_ADDR 0x4030CE38     /* IRQ exp handler address      */

#define MAXADDR 0xA0000000      /* 512 MB RAM starting from 0x80000000  */
```

22.6 Xinu System Startup

As the last step, the code in start.S calls function *nulluser*. A single program, not an operating system, is running when the call occurs. *Nulluser* initializes the operating system, creates a process to execute function *main*, and becomes the null process. If there is drama in the system, it lies here, where the transformation from program to concurrent system occurs. The code is found in file *initialize.c*.

```
/* initialize.c - nulluser, sysinit, sizmem */

/* Handle system initialization and become the null process */

#include <xinu.h>
#include <string.h>

extern  void    start(void);    /* Start of Xinu code                   */
extern  void    *_end;          /* End of Xinu code                     */

/* Function prototypes */

extern  void main(void);        /* Main is the first process created    */
extern  void xdone(void);       /* System "shutdown" procedure          */
static  void sysinit();         /* Internal system initialization       */
extern  void meminit(void);     /* Initializes the free memory list     */

/* Declarations of major kernel variables */

struct  procent proctab[NPROC]; /* Process table                        */
struct  sentry  semtab[NSEM];   /* Semaphore table                      */
struct  memblk  memlist;        /* List of free memory blocks           */

/* Active system status */

int     prcount;                /* Total number of live processes       */
pid32   currpid;                /* ID of currently executing process    */

/*------------------------------------------------------------------------
 * nulluser - initialize the system and become the null process
 *
 * Note: execution begins here after the C run-time environment has been
 * established.  Interrupts are initially DISABLED, and must eventually
 * be enabled explicitly.  The code turns itself into the null process
 * after initialization.  Because it must always remain ready to execute,
```

```
 * the null process cannot execute code that might cause it to be
 * suspended, wait for a semaphore, put to sleep, or exit.  In
 * particular, the code must not perform I/O except for polled versions
 * such as kprintf.
 *------------------------------------------------------------------------
 */

void    nulluser()
{
        struct  memblk  *memptr;        /* Ptr to memory block          */
        uint32  free_mem;               /* Total amount of free memory  */

        /* Initialize the system */

        sysinit();

        kprintf("\n\r%s\n\n\r", VERSION);

        /* Output Xinu memory layout */
        free_mem = 0;
        for (memptr = memlist.mnext; memptr != NULL;
                                            memptr = memptr->mnext) {
                free_mem += memptr->mlength;
        }
        kprintf("%10d bytes of free memory.  Free list:\n", free_mem);
        for (memptr=memlist.mnext; memptr!=NULL;memptr = memptr->mnext) {
            kprintf("           [0x%08X to 0x%08X]\r\n",
                (uint32)memptr, ((uint32)memptr) + memptr->mlength - 1);
        }

        kprintf("%10d bytes of Xinu code.\n",
                (uint32)&etext - (uint32)&text);
        kprintf("           [0x%08X to 0x%08X]\n",
                (uint32)&text, (uint32)&etext - 1);
        kprintf("%10d bytes of data.\n",
                (uint32)&ebss - (uint32)&data);
        kprintf("           [0x%08X to 0x%08X]\n\n",
                (uint32)&data, (uint32)&ebss - 1);

        /* Enable interrupts */

        enable();

        /* Create a process to execute function main() */
```

```
        resume (
           create((void *)main, INITSTK, INITPRIO, "Main process", 0,
           NULL));

        /* Become the Null process (i.e., guarantee that the CPU has    */
        /*  something to run when no other process is ready to execute) */

        while (TRUE) {
                ;               /* Do nothing */
        }

}

/*------------------------------------------------------------------------
 *
 * sysinit  -  Initialize all Xinu data structures and devices
 *
 *------------------------------------------------------------------------
 */
static  void    sysinit()
{
        int32   i;
        struct  procent *prptr;         /* Ptr to process table entry   */
        struct  sentry  *semptr;        /* Ptr to semaphore table entry */

        /* Platform Specific Initialization */

        platinit();

        /* Initialize the interrupt vectors */

        initevec();

        /* Initialize free memory list */

        meminit();

        /* Initialize system variables */

        /* Count the Null process as the first process in the system */

        prcount = 1;

        /* Scheduling is not currently blocked */

        Defer.ndefers = 0;
```

```
/* Initialize process table entries free */

for (i = 0; i < NPROC; i++) {
        prptr = &proctab[i];
        prptr->prstate = PR_FREE;
        prptr->prname[0] = NULLCH;
        prptr->prstkbase = NULL;
        prptr->prprio = 0;
}

/* Initialize the Null process entry */

prptr = &proctab[NULLPROC];
prptr->prstate = PR_CURR;
prptr->prprio = 0;
strncpy(prptr->prname, "prnull", 7);
prptr->prstkbase = getstk(NULLSTK);
prptr->prstklen = NULLSTK;
prptr->prstkptr = 0;
currpid = NULLPROC;

/* Initialize semaphores */

for (i = 0; i < NSEM; i++) {
        semptr = &semtab[i];
        semptr->sstate = S_FREE;
        semptr->scount = 0;
        semptr->squeue = newqueue();
}

/* Initialize buffer pools */

bufinit();

/* Create a ready list for processes */

readylist = newqueue();

/* Initialize the real time clock */

clkinit();

for (i = 0; i < NDEVS; i++) {
        init(i);
}
```

```
        return;
}

int32   stop(char *s)
{
        kprintf("%s\n", s);
        kprintf("looping... press reset\n");
        while(1)
                /* Empty */;
}

int32   delay(int n)
{
        DELAY(n);
        return OK;
}
```

Nulluser itself is exceedingly simple. It calls function *sysinit* to initialize the system data structures. When *sysinit* returns, the running program has become the null process (process 0), but interrupts remain disabled and no other processes exist. After printing a few introductory messages, *nulluser* enables interrupts, and calls *create* to start a process running the user's main program.

Because the program executing *nulluser* has become the null process, it cannot exit, sleep, wait for a semaphore, or suspend itself. Fortunately, the initialization function does not perform any action that takes the caller out of the current or ready states. If such actions were needed, *sysinit* would have created another process to handle the action. Once initialization is complete and a process has been created to execute the user's main program, the null process falls into an infinite loop, giving *resched* a process to schedule when no user processes are ready to run.

22.7 Transforming A Program Into A Process

Function *sysinit* handles the task of system initialization. It makes a series of calls to initialize the hardware platform (*platinit*), exception vectors (*initevec*), and the memory free list (*meminit*). It initializes remaining system data structures, such as the process table and semaphore table, and calls *clkinit* to initialize the real-time clock. Finally, *sysinit* iterates through the devices that have been configured and calls the initialization function for each device. To do so, it invokes *init* for the device ID.

The most interesting piece of the initialization code occurs about half-way through *sysinit* when it fills in the process table entry for process zero. Many of the process table fields, such as the process name field, can be left uninitialized — they are filled in merely to make debugging easier. The real work is done by the two lines that set the current process ID variable, *currpid*, to the ID of the null process, and assign *PR_CURR*

to the process state field of the process table entry. Until *currpid* and the state have been assigned, rescheduling is impossible. Once the process table entry and *currpid* have been assigned, the program has become a currently running process that has process ID zero, and is available for context switching.

To summarize:

> *After it fills in the process table entry for process zero, the code sets variable* currpid *to zero, which transforms the sequential program into a process.*

Once the null process has been created, all that remains is for *sysinit* to initialize the other pieces of the system before it returns so that all services are available when function *nulluser* starts a process executing the user's main program. When it is resumed, the new process will have a higher priority than the null process. Therefore, *resched* will switch context to the new process, and the scheduling invariant will be in effect.

22.8 Perspective

The intellectually significant aspects of operating system design arise when new abstractions are created on top of low-level hardware. In the case of system initialization, the details are not as important as the conceptual transformation: the processor starts a fetch-execute cycle running instructions sequentially, and the initialization code self-transforms into a concurrent processing system. The important point is that the initial code does not create a separate, concurrent system and then jump to the new system. No leap is made to reach the abstraction, and the original sequential execution is not abandoned. Instead, the running code declares itself to be a process, fills in the system data structures needed to support the declaration, and eventually allows other processes to execute. Meanwhile, the processor continues the fetch-execute cycle, and the new abstraction emerges without any disruption.

22.9 Summary

Initialization is the last step of system design; it should be postponed to avoid changing the design simply to make initialization easier. Although initialization involves many details, the most conceptually interesting part involves the transformation from a sequential program to a system that supports concurrent processing. To make itself correspond to the null process, the code fills in the process table entry for process zero and sets variable *currpid* to zero.

EXERCISES

22.1 What is the BIOS on an x86 computer, and what does it do?

22.2 If you were designing a bootstrap loader program, what functionality would you include? Why?

22.3 Read about the *GRUB* and *UBOOT* bootstrap programs. What functionality does each provide?

22.4 Is the order of initialization important for the process table, semaphore table, memory free list, devices, and ready list? Explain.

22.5 On many systems, it is possible to create a function *sizmem* that finds the highest valid memory address by probing memory until an exception occurs. Is such a function possible on the example platforms? Why or why not?

22.6 Explain, by tracing through the functions involved, what would go wrong if *nulluser* enabled interrupts *before* calling *sysinit*.

22.7 The network code, remote disk driver, and remote file system driver each create a process. Should the processes be created in *sysinit*? Why or why not?

22.8 Most operating systems arrange for the network code to run and obtain an IP address before the system starts any user processes. Design a way for Xinu to create a network process, wait for the network process to obtain an IP address, and then create a process to run the main program. Be careful: the null process cannot block.

Chapter Contents

23.1 Introduction, 589
23.2 Self-initializing Modules, 590
23.3 Self-initializing Modules In A Concurrent System, 591
23.4 Self-initialization In The Presence Of Reboot, 593
23.5 Initialization Using Accession Numbers, 593
23.6 A Generalized Memory Marking Scheme, 595
23.7 Data Declarations For The Memory Marking System, 596
23.8 Implementation Of Marking, 598
23.9 Perspective, 599
23.10 Summary, 599

23

Subsystem Initialization And Memory Marking

The best thing about endings is knowing that just ahead lies the daunting task to start over

— Jodi Picoult

23.1 Introduction

The previous chapter considers hardware bootstrap and operating system initialization. The chapter explains how an operating system starts as a program that executes instructions, and how, after initializing system variables and data structures, the initialization code creates a process to start executing the *main* function, and then becomes the null process.

This chapter extends our examination of initialization by considering two broad questions. First, if we divide sets of functions into modules, how can each perform self-initialization without relying on a central initialization function? Second, if modules remain resident in memory across system reboots, how can each module know whether it has been reinitialized following the latest reboot? The chapter introduces an elegant, compact mechanism called *memory marking* that allows self-initializing modules to be created. The system correctly identifies whether a module needs to be reinitialized, even if the module remains resident in memory across a system reboot.

23.2 Self-initializing Modules

We use the term *module* to refer to a set of functions and the shared data on which they operate. Although the C language does not have encapsulation mechanisms that permit a programmer to specify modules, we can think of code being divided into conceptual modules. For example, consider the buffer pool mechanism described in Chapter 10 and the high-level message passing mechanism described in Chapter 11. In each case, a set of functions operates on a shared data structure. The buffer pool module offers four primary functions:

poolinit

mkpool

getbuf

freebuf

Each of the functions is written to assume that *poolinit* will be called before any of the other functions are invoked. As we have seen, *sysinit* makes the call to *poolinit* when the operating system functions are being initialized. Similarly, *sysinit* calls *ptinit* to initialize the port mechanism.

Relying on *sysinit* to initialize each module has several disadvantages. First, a programmer cannot add a new module to the operating system without modifying a fundamental system function, *sysinit*. More important, a programmer must understand which other modules a given initialization function uses, and must insert the new initialization call after the other modules are initialized. Second, if the initialization call is accidentally omitted, the module may operate incorrectly and cause the system to fail. Third, placing a call of the initialization function in *sysinit* forces the loader to include the initialization function and shared data items, even if other functions in the module are not used. On an embedded system with constrained memory, programmers cannot afford to include items that are not needed — if *sysinit* references all possible modules, the image may be too large to fit in memory.

How can a programmer arrange for module initialization without placing an explicit call in *sysinit*? The answer lies in *self-initializing modules*. For a conventional C program, self-initializing is trivial: declare a global variable with an initial value that specifies whether initialization has been performed, and insert code at the beginning of each function to test the global variable. For example, Figure 23.1 shows the general structure of code for a module that has two functions, *func_1* and *func_2*, plus an initialization function, *func_init*. Each function includes an extra line of code that examines global variable *needinit*, which is statically initialized to 1. If *needinit* is 1 when a function is called, *func_init* is invoked to perform initialization.

```
/* Example of a self-initializing module that uses a global variable */

int32 needinit = 1;                 /* Non-zero until initialized    */
...declarations for other global data structures

void  func_1(...args) {

        if (needinit) func_init();  /* Initialize before proceeding  */
        ...code for func_1
        return;
}

void  func_2(...args) {

        if (needinit) func_init();  /* Initialize before proceeding  */
        ...code for func_2
        return;
}

void  func_init(void) {

        if (needinit != 0) {        /* Initialization is needed      */
              ...code to perform initialization
              needinit = 0;
        }
        return;
}
```

Figure 23.1 An example showing the structure of a conventional C program that uses a global variable to control self-initialization.

Although it works in a conventional program, the approach outlined above does not suffice for modules in our operating system. There are two reasons: concurrency and reboot. First we will discuss how to handle concurrency, and then we will look at reboot.

23.3 Self-initializing Modules In A Concurrent System

To see what can happen to the module in Figure 23.1 if concurrent processes make calls to the functions, consider a worst-case scenario. Suppose two processes, P_1 and P_2 attempt to use the module, and consider context switching between them. If one process calls function *func_1* and the other calls *func_2*, both processes could be executing *func_init* to initialize global data structures the module uses. One of the processes

might finish initialization, return to its function, and start using the global data structures while the other process continues initialization which overwrites the values.

The options for handling concurrent execution consist of disabling interrupts or using a mutual exclusion semaphore. A mutex semaphore minimizes interference with other system activities — a process will only block if it attempts to use the module while another process is already using it. Otherwise, no processes block, and interrupts are not disabled. However, semaphore creation requires initialization, which means we cannot use semaphores to control access to the initialization function. Figure 23.2 illustrates a hybrid solution that disables interrupts during initialization, and uses a semaphore to provide mutual exclusion for normal execution.

```
/* A self-initializing module that permits concurrent access        */

int32 needinit = 1;                 /* Non-zero until initialized    */
sid32 mutex;                        /* Mutual exclusion semaphore ID */

void  func_1(...args) {
        intmask mask;

        mask = disable();           /* Disable during initialization */
        if (needinit) func_init();  /* Initialize before proceeding  */
        restore(mask);              /* Restore interrupts            */
        wait(mutex);                /* Use mutex for exclusive access*/
        ...code for func_1
        signal(mutex);              /* Release the mutex             */
        return;
}
...other functions in the module structured as above
void  func_init(void) {
        intmask mask;

        mask = disable();
        if (needinit != 0) {        /* Initialization is still needed*/
              mutex = semcreate(1); /* Create the mutex semaphore    */
              ...code to perform other initialization
              needinit = 0;
        }
        restore(mask);
        return;
}
```

Figure 23.2 An illustration of a self-initializing module that permits concurrent access.

As the code shows, *disable* and *restore* are only used when the initialization is tested or performed, which means functions in the module can take arbitrarily long without causing problems for devices. Note that *func_init* disables interrupts even though individual functions (e.g., *func_1*) disable interrupts before calling *func_init*. The use of *disable* and *restore* allows *func_init* to be called directly by a user process or from a function in the module. In either case, the initialization function will return to the caller with interrupts in the same state as when the call was made. Exercises consider alternative structures.

Also note that when the initialization finishes, two things will have been accomplished. First, the module's data structures will have been initialized. Second, the mutual exclusion semaphore will have been created, and its ID will have been placed in global variable *mutex* ready for functions in the module to use.

23.4 Self-initialization In The Presence Of Reboot

As an added complication, consider rebooting a system where all code and data remain resident in memory. For example, an early version of Xinu allowed reboot to avoid slow speed downloading. Rebooting a resident system adds a complication to initialization because global variables (i.e., variables in the data segment) will not be reset to their initial values. For example, consider variable *needinit* in Figure 23.2. When the operating system is loaded, the variable has the value 1. As soon as any of the functions in the module are called, initialization will be performed, and *needinit* will have the value 0. If the operating system is restarted, the module will not be reinitialized.

To understand the potential consequences, consider a module that allocates heap memory. The initialization function will call *getmem*, and will save a pointer to the allocated memory. Functions in the module use the pointer to access the allocated memory. When the operating system reboots, all heap memory will be placed on the free list, and will be available for allocation. Suppose that after the reboot, the order in which processes execute differs from the order in the initial run. It could happen that the memory block allocated to a module in the first run of the system is allocated for another purpose in the second run. If a module continues to assume it owns the memory block that was allocated in the first run, the module can write into a memory block owned by another process. Such errors are especially difficult to find.

23.5 Initialization Using Accession Numbers

For systems where the integer size is large and reboots do not occur often, the initialization problem can be handled with an *accession number* scheme. The scheme uses a single global counter for the operating system plus a global counter for each module. The operating system counter is used to count how many times the operating system has rebooted, and the global variable for each module is used to count the number of times

the module has been initialized. The value of all counters is initialized to 0 at compile time. The operating system boot counter is incremented when *sysinit* executes, so it will be 1 on the first run of the system.

Assuming a module must be initialized once each time the system boots, the use of accession numbers should be clear. The module initialization test can be replaced by a comparison: the module should be initialized if the system's boot counter exceeds the module's initialization counter. Figure 23.3 illustrates the use of accession numbers.

```
/* A self-initializing module using accession numbers                */


extern int32  boot;                /* Count of times OS boots        */
int32  minit = 0;                  /* Count of module initializations*/
sid32  mutex;                      /* Mutual exclusion semaphore ID  */

void  func_1(...args) {

       if (minit<boot) func_init();/* Initialize before proceeding   */
       wait(mutex);                /* Use mutex for exclusive access */
       ...code for func_1
       signal(mutex);              /* Release the mutex              */
       return;
}
...other functions in the module structured as above

void  func_init(void) {
       intmask mask;

       mask = disable();
       if (minit < boot) {         /* Initialization is still needed */
               mutex = semcreate(1); /* Create the mutex semaphore   */
               ...code to perform other initialization
               minit++;
       }
       restore(mask);
       return;
}
```

Figure 23.3 Illustration of a self-initializing module using accession numbers to handle system reboot.

23.6 A Generalized Memory Marking Scheme

The accession number scheme illustrated in Figure 23.3 makes an important assumption: the reboot counter will never wrap around. On a system with 32-bit integers, rebooting once per second and using unsigned arithmetic means the system can run for over one-hundred years before the counter wraps around. On 8-bit and 16-bit embedded processors, an accession number mechanism will not suffice.

The question arises: how can self-initializing modules work in a concurrent environment on a processor with a small range of integer values? Of course, one could write code to compute 32-bit values using smaller integers. However, the author created an elegant and efficient technique to handle the problem. Known as *memory marking*, the technique requires almost no overhead and accommodates dynamic module loading as well as the static modules described in this chapter.

Memory marking defines a new data type (*memmark*) and two functions to manipulate the data type:

```
memmark  L;   – Declare L to be a location that can be marked
mark(L);      – Mark location L
notmarked(L)  – Non-zero if location L is not marked
```

After an operating system reboot, none of the locations in memory has been marked. Thererfore, *notmarked* returns non-zero. After *mark* has been called to mark a location, *notmarked* returns 0 for that location. It may seem odd that the second function tests whether a location is unmarked instead of whether the location is marked. However, we will see that the most common situation involves taking action when a location has not yet been marked. For example, the code for a typical check might appear:

```
if ( notmarked(xxx) ) {
      ...code to perform initialization
      mark(xxx);
}
```

where *xxx* is a variable declared to be of type memmark. Of course, when used in a concurrent processing system, additional code is needed to disable interrupts to guarantee that no other process can interfere with initialization.

Figure 23.4 shows how memory marking facilities can be inserted into a self-initializing module.

```
/* A self-initializing module using memory marking                    */

memmark loc;                        /* Count of module initializations*/
sid32   mutex;                      /* Mutual exclusion semaphore ID  */

void  func_1(...args) {

        if (notmarked(loc)) func_init(); /* Initialize the module    */
        wait(mutex);                /* Use mutex for exclusive access */
        ...code for func_1
        signal(mutex);              /* Release the mutex              */
        return;
}
...other functions in the module structured as above

void  func_init(void) {
        intmask mask;

        mask = disable();
        if (notmarked(loc)) {
               mutex = semcreate(1); /* Create the mutex semaphore   */
               ...code to perform other initialization
               mark(loc);
        }
        restore(mask);
        return;
}
```

Figure 23.4 An illustration of a module using memory marking.

23.7 Data Declarations For The Memory Marking System

The implementation of memory marking is surprisingly efficient in both the time required and space occupied. In essence, the system stores a list of memory locations that have been marked in array *marks*. The system uses integer *nmarks* to store an integer count of the number of items currently in the *marks* array. Each marked location is an integer. When the location is marked, the system stores a value in the location that gives the index in the *marks* array corresponding to the location.

Testing whether a location L has been marked is straightforward: extract the integer value from location L, call it I. If I is out of range of the *marks* array or the Ith

entry in the *marks* array does not contain *L*, the location is not marked. File *mark.h* contains the data declarations as well as inline function *notmarked*.

```
/* mark.h - notmarked */

#define MAXMARK 20              /* Maximum number of marked locations   */

extern  int32   *(marks[]);
extern  int32   nmarks;
extern  sid32   mkmutex;
typedef int32   memmark[1];     /* Declare a memory mark to be an array */
                                /*   so user can reference the name     */
                                /*   without a leading &                */

/*------------------------------------------------------------------------
 *  notmarked  -  Return nonzero if a location has not been marked
 *------------------------------------------------------------------------
 */
#define notmarked(L)            (L[0]<0 || L[0]>=nmarks || marks[L[0]]!=L)
```

The memory marking code exploits a C language subtlety to make using memory marks easier and safer. In C, the name of an array gives a pointer to the array. When a programmer declares a variable to be of type *memmark*, the result is an integer array with one element. For example, when a programmer declares:

```
memmark  loc;
```

the compiler allocates storage for a single integer. However, because the type of variable *loc* is an array rather than an integer, references to *loc* in a program result in a pointer. Therefore, when a programmer writes:

```
mark(loc);
```

the compiler passes the address of *loc* without requiring a programmer to remember to include an ampersand. The code for *notmarked* should now be clear: when a programmer writes:

```
notmarked(loc)
```

to invoke the *notmarked* macro, argument *loc* will be an array with one location, not an integer. Therefore, to find the integer value stored in *loc*, the code dereferences the array (i.e., refers to *loc[0]* rather than to *loc*).

23.8 Implementation Of Marking

File *mark.c* contains code for the two remaining functions of the memory marking system: *markinit* and *mark*. Function *markinit* is called when the operating system reboots (e.g., in Xinu, it is called from *sysinit*). *Markinit* sets the count of marked locations to zero, and allocates a mutual exclusion semaphore. Once *markinit* has initialized the system, modules can call *mark* to mark a specific location.

```
/* mark.c - markinit, mark */

#include <xinu.h>

int32   *marks[MAXMARK];                /* Pointers to marked locations */
int32   nmarks;                         /* Number of marked locations   */
sid32   mkmutex;                        /* Mutual exclusion semaphore   */

/*------------------------------------------------------------------------
 *  markinit  -  Called once at system startup
 *------------------------------------------------------------------------
 */
void    markinit(void)
{
        nmarks = 0;
        mkmutex = semcreate(1);
}


/*------------------------------------------------------------------------
 *  mark  -  Mark a specified memory location
 *------------------------------------------------------------------------
 */
status  mark(
          int32 *loc                    /* Location to mark             */
        )
{

        /* If location is already marked, do nothing */

        if ( (*loc>=0) && (*loc<nmarks) && (marks[*loc]==loc) ) {
                return OK;
        }

        /* If no more memory marks are available, indicate an error */
```

```
        if (nmarks >= MAXMARK) {
                return SYSERR;
        }

        /* Obtain exclusive access and mark the specified location */

        wait(mkmutex);
        marks[ (*loc) = nmarks++ ] = loc;
        signal(mkmutex);
        return OK;
}
```

One line of code in function *mark* does the work of marking a location:

```
marks[ (*loc) = nmarks++ ] = loc;
```

The single line increments the number of marks, assigns the original value to the location, and uses the number as an index into the *marks* array to set the entry to the address of the location being marked.

23.9 Perspective

Since the publication of the first edition of the Xinu book, engineers have written to say they used the memory marking approach when building embedded software systems. In other words, the mechanism is quite practical. The author originally created the memory marking mechanism while he was a graduate student, and the purpose was to solve a homework problem in a theory class. Ironically, the professor in the class did not appreciate the practicality of the scheme.

Even if you skim over everything else in this chapter, take a moment to look at the line of code discussed above. In most languages, marking a memory location would require multiple lines of code, and would not compile into minimal instructions. In C, the source code is incredibly compact, and the compiler can produce an efficient implementation.

23.10 Summary

Several techniques are available to allow a module to initialize itself. The standard programming technique of using an initialized global variable can be extended to a concurrent programming environment by disabling interrupts during initialization. A hybrid scheme can restrict disabling interrupts to the initialization code, and can use a mutual exclusion semaphore to allow the functions in the module to execute with interrupts enabled.

The memory marking technique generalizes self-initialization to handle reboot of an operating system without reloading the image. Memory marking is both efficient and elegant. The implementation uses the trick of declaring an array of one location to allow a programmer to declare a memory mark, *x*, pass the name *x* as an argument to *notmarked* and *mark*, even though the functions require an address.

EXERCISES

23.1 The module shown in Figure 23.2 arranges for function *func_1* to disable interrupts before the *if* statement that checks global variable *needinit*, and restore interrupts afterward. Argue that removing the calls to *disable* and *restore* in *func_1* will not change correctness.

23.2 Extend the previous exercise by removing the calls to *disable*, *restore*, and the conditional check of *needinit* (i.e., simply have *func_1* call *func_init* and then wait on the mutex). Show that the resulting code is still correct. What is the advantage of keeping the *if* statement?

23.3 Can multiple processes call *notmarked* and *mark* concurrently or will problems result? Explain.

23.4 The code in *mark* can be rewritten to use the *notmarked* macro. What is the disadvantage of using the macro?

23.5 Modify the buffer pool functions from Chapter 10 to use memory marking and to be self-initializing.

23.6 Consider an operating system that provides dynamic loading (i.e., the system can load a module into memory at runtime). Is memory marking useful in such a system? Explain.

Chapter Contents

24.1 Introduction, 603
24.2 Terminology: Faults, Checks, Traps, And Exceptions, 603
24.3 Vectored Exceptions And Maskable Interrupts, 604
24.4 Types Of Exceptions, 604
24.5 Handling Exceptions, 605
24.6 Exception Vector Initialization, 606
24.7 Panic In The Face Of Catastrophic Problems, 606
24.8 Implementation Of Panic, 607
24.9 Perspective, 607
24.10 Summary, 608

24

Exception Handling

I never make exceptions. An exception disproves the rule.

— Sir Arthur Conan Doyle

24.1 Introduction

This chapter discusses the topic of exception handling. Because the underlying hardware dictates exactly how an exception is reported, the techniques an operating system uses to handle exceptions depend entirely on the hardware. Therefore, we will describe how exceptions are handled on the example platforms, and leave readers to explore other systems and architectures.

In general, exception handling is concerned with details rather than concepts. Therefore, the chapter contains fewer broad concepts than earlier chapters. The reader should think of it as an example, and be aware that both the details and techniques may change if another hardware system is used.

24.2 Terminology: Faults, Checks, Traps, And Exceptions

A variety of terms have been used to describe problems that can occur during execution, and various hardware vendors have preferred terminology. Early computers used the term *check* (or *machine check*) to refer to an internal hardware malfunction. For example, an early version of Xinu ran on hardware that could detect a power supply failure. That is, as the voltage began to fall, the processor informed the operating system.†

†Although it seems unlikely that an operating system would have time to respond during a power failure, on the occasion when a power supply did fail, Xinu was able to print a "power failure" message before losing power.

Originally, the term *fault* was used to refer to problems associated with hardware, as opposed to errors that are caused by software. When demand paging was invented, however, the term *page fault* was used to describe a condition where a referenced page is not resident in memory. As paging became popular, faults became associated with page faults, and the distinction between hardware faults and software faults was lost.

When an error situation arises, the hardware must have a way to inform the operating system. Some vendors use the term *trap*, and say that the hardware traps to the operating system. As we have seen, the hardware must also inform the operating system when an I/O device needs service, and hardware is often designed to use a single mechanism for both errors and interrupts.

24.3 Vectored Exceptions And Maskable Interrupts

Currently, most vendors use the generic term *exception* to refer to any condition that is unexpected. We say that the hardware "raises an exception" to indicate a problem. Exceptions include a broad range of conditions, including page faults, arithmetic errors (e.g., divide by zero), illegal instructions, bus errors, page faults, and I/O interrupts. As we have seen, systems can use a single vectored mechanism to handle all exceptions.

Despite using a single mechanism, all exceptions are not treated exactly the same. The main difference arises because a processor can *mask* some exceptions, but not others. We have seen that an operating system can disable interrupts. The term *maskable interrupt* occurs because disabling interrupts requires the operating system to set *interrupt mask* bits in an internal hardware register. Some hardware only provides one mask bit (i.e., interrupts are disabled or enabled). Other hardware uses multiple bits to provide multiple levels of interrupts. On such systems, a processor can disable interrupts on lower-priority devices while still permitting interrupts on higher-priority devices. Interrupt controller hardware may also provide a mask for specific IRQs, which allows a processor to disable or enable interrupts on individual devices (e.g., to stop a Wi-Fi device without affecting other devices).

24.4 Types Of Exceptions

Exceptions can be grouped into seven key types:

- Device interrupts
- Arithmetic exceptions
- Illegal memory references
- I/O errors (bus errors)
- Protection faults
- Invalid instructions
- Hardware failures

Earlier chapters examine device interrupts in detail, and point out that a misconfigured device can generate an interrupt with an invalid IRQ. Arithmetic exceptions include attempts to divide by zero, and floating point overflow/underflow. On hardware in which the memory system is separate from other I/O, the hardware will raise a memory exception if software references an address that does not correspond to physical memory. On RISC computers where memory references must be aligned, a memory error can result if software attempts to fetch or store an integer from an address that is not a multiple of the word size. Some hardware does not distinguish between illegal memory references and I/O errors — a single exception type (bus error) is used for both. I/O errors do not always mean that device hardware has malfunctioned. It is possible for an I/O error to occur if a device driver passes an invalid address to a device. For example, if DMA hardware is using a descriptor ring in memory and the ring contains an invalid pointer, the DMA hardware will cause an error by attempting to use the pointer.

Protection faults occur when software attempts to perform an operation that is not permitted at the current level of protection. For example, the hardware raises a protection fault if an application attempts to execute an instruction that requires operating system privilege.

Invalid instructions are rare because most code is generated by a compiler. However, if an error causes a program to branch to an arbitrary location in memory, the location may not contain an instruction. Therefore, programmers must be especially careful when writing code, such as the device-independent I/O functions, that uses indirect function calls.

With current technology, hardware failures are also rare. Solid-state electronics has made hardware extremely reliable. However, battery-powered devices introduce a specific failure mode because a battery can lose voltage gradually as the charge is drained, and hardware can misbehave if the voltage slips below a minimum threshold.

24.5 Handling Exceptions

When an unexpected interrupt occurs, the solution is straightforward: ignore the interrupt (or if a new device has been added, reconfigure the system to include the device). Exception processing is more complex, and depends on the size and purpose of the system as well as the source of the exception. In particular, operating systems must be correct and reliable. Therefore, if operating system code causes an exception, the problem is serious, and may warrant stopping the system.

When application code causes an exception, the problem is less serious and is assumed to be local to the application. Two general approaches have been used to handle application exceptions: termination and notification. Termination means the process that caused the exception is killed. Notification means the operating system calls an exception handler associated with the application process. That is, when it begins execution, an application informs the operating system of a function to call for exceptions that it wishes to catch. When an exception occurs, the operating checks the stored in-

formation to see whether the process registered an exception handler for the exception. If so, the operating system calls the exception handler, and then allows the process to continue. Some programming languages have a way to declare exception handlers.

For an embedded system, recovery from an exception is usually difficult or impossible. Even if the system allows interaction with a user, the user can do little to correct the problem. Thus, many embedded systems either reboot or power down when an exception occurs.

24.6 Exception Vector Initialization

We said that although some exceptions are maskable, many are not. The hardware uses the exception to choose an entry in the vector, and assumes the entry contains the address of a handler. Therefore, an operating system designer must be careful to ensure that no exceptions can occur until the system fills in the exception vector.

Arranging an operating system to associate exceptions with the appropriate handler, filling in vectors, and building dispatch code that invokes a handler are tedious tasks that plague implementers. The mismatch between low-level hardware and high-level operating system abstractions makes it difficult to map exceptions onto the code that caused the exception. For example, suppose a device using DMA is following pointers on a descriptor ring in memory, and encounters an invalid pointer. The hardware raises an exception, but the exception has nothing to do with the application that is currently executing. Thus, it may be difficult for an operating system designer to identify the correct source of a problem.

24.7 Panic In The Face Of Catastrophic Problems

Our example code follows the Unix tradition, and uses the name *panic* for the function that handles catastrophic problems. *Panic* is only invoked in situations where processing cannot continue. The idea is straightforward: *panic* takes a string as an argument, displays the string on the console, and then halts the processor. The code is minimal: *panic* does not attempt to recover, nor does it attempt to identify the offending process.

Because many hardware-specific details are involved, a version of *panic* that displays registers or processor state may need to be written in assembly language. For example, an exception can occur because a stack pointer is invalid, and a *panic* function may need to avoid using the stack. Consequently, to work under all circumstances, *panic* code cannot merely attempt to push a value on the stack or execute a function call. Similarly, because entries in the device switch table may be incorrect, a *panic* function that extracts information about the *CONSOLE* device from the device switch table may not work. Fortunately, most of these cases are extreme. Therefore, many operating system designers start with a basic version of the *panic* function that works as long as the bulk of the operating system and runtime environment remain intact.

24.8 Implementation Of Panic

Our version of a *panic* function is simplistic. Because interrupt processing may have caused the exception, *panic* begins by disabling further interrupts, and then uses polled I/O to display the message on the console (i.e., it uses *kprintf*). To stop the processor, the code merely enters a tight loop; the exercises suggest alternatives.

File *panic.c* contains the code:

```
/* panic.c - panic */

#include <xinu.h>

/*------------------------------------------------------------------------
 * panic  -  Display a message and stop all processing
 *------------------------------------------------------------------------
 */
void    panic (
         char   *msg                    /* Message to display           */
        )
{
        disable();                      /* Disable interrupts           */
        kprintf("\n\n\rpanic: %s\n\n", msg);
        while(TRUE) {;}                 /* Busy loop forever            */
}
```

24.9 Perspective

The question of how to handle exceptions is more complex than it may seem. To see why, consider what happens after an application makes a system call: although it remains running, the application process executes operating system code. If an exception occurs, the exception should be considered an operating system problem and should not invoke the exception handler for the process. Similarly, if an exception occurs while an application is executing code from a shared library, the exception should not be treated differently than an exception caused by the application. Such distinctions require the operating system to keep track of exactly what an application is currently doing.

Another type of complexity arises because exceptions can be caused by the interaction of processes. For example, if a Xinu process inadvertently writes into another process's address space, the second process may experience an exception that the first process caused. Thus, even if the system provides a mechanism to catch exceptions, the exception handler for the second process may not anticipate the problem, and may have no way to recover.

24.10 Summary

Trapping and identifying exceptions and unexpected interrupts are important because they help isolate bugs that arise as an operating system is being implemented. Hence, building error detection functions early is essential, even if the implementation is crude and unsophisticated.

In embedded systems, an exception usually causes the system to reboot or power down. Our example implementation of *panic* does not assume interrupts are working, and does not attempt to use operating system functions. Instead, the code disables interrupts, prints a message on the console, and enters a hard loop to stop further processing.

EXERCISES

24.1 Rewrite Xinu to make the code serially reusable, and modify *panic* to wait 15 seconds and then jump back to the starting location (i.e., reboot Xinu).

24.2 How many locations does *panic* require on the runtime stack to process an exception?

24.3 Design a mechanism that allows an executing process to catch exceptions.

24.4 Some processors include a *halt* instruction that shuts down the processor. Find example systems that have such an instruction, and rewrite the code.

24.5 Find a situation where an error can occur, but *panic* cannot be called. Hint: what happens during a function call?

Chapter Contents

25.1 Introduction, 611
25.2 The Need For Multiple Configurations, 611
25.3 Configuration In Xinu, 613
25.4 Contents Of The Xinu Configuration File, 613
25.5 Computation Of Minor Device Numbers, 616
25.6 Steps In Configuring A Xinu System, 616
25.7 Perspective, 617
25.8 Summary, 617

25

System Configuration

No pleasure endures unseasoned by variety.

— Publilius Syrus

25.1 Introduction

This chapter concludes the discussion of basic operating system design by answering a practical question: how can the code from earlier chapters be transformed to make it suitable for a given computer that has a specific set of peripheral devices?

The chapter discusses the motivation for configuration, tradeoffs between static and dynamic configuration, and presents a basic configuration program that takes a description of the system and generates source files tailored to the description.

25.2 The Need For Multiple Configurations

The earliest computers were designed as monolithic systems, meaning that the hardware and software were designed together. A designer chose the details of the processor, memory, and I/O devices, and an operating system was built to control the specific hardware that had been selected. Later generations of computers added options, allowing a customer to choose between a large or small memory and a large or small disk. As the industry matured, third-party vendors began selling peripheral devices that could be attached to a computer. Current computer users have many possibilities — an owner can purchase hardware devices from a variety of vendors. Thus, a given computer can have a combination of hardware devices unlike other computers.

Two broad approaches have been used to configure operating system software:

- Static configuration
- Dynamic configuration

Static configuration. Static configuration is appropriate for small, "self-contained" systems, where the hardware does not change. A designer chooses the hardware, including processor, memory, and a set of peripheral devices. Once the hardware has been specified, an operating system is created that supports exactly the hardware that has been chosen, without any extra software modules. In practice, one does not create a new operating system for each hardware design. Instead, a general-purpose operating system is used, and a subset of the system modules are selected. Usually, the hardware specification becomes input to a *configuration program* that manages operating system source code. The configuration program uses the hardware specification to select the modules that are needed for the target hardware, and excludes other modules. When the resulting code has been compiled and linked, we say it has been *configured* for the hardware.

Dynamic configuration. The alternative to static configuration is dynamic configuration in which parts of the operating system can be changed while the system executes. Dynamic configuration is only appropriate for systems that have large memories and secondary storage. A basic operating system starts running without an exact knowledge of the hardware. The system probes the hardware, determines which devices are present, and loads pieces of the system for the hardware. Of course, the pieces of operating system software must be available on a local disk or downloaded (e.g., over the Internet).

Static configuration is a form of early binding. The chief advantage is that the memory image only contains modules for the hardware that exists. Static configuration is ideal for the smallest embedded systems where memory is limited and no secondary storage is available. Another advantage arises because the system does not spend time identifying hardware during the bootstrap process; the information is bound into the code when the operating system image is created. Thus, a system can boot instantly. The chief disadvantage of early configuration is that a system configured for one machine cannot run on another unless the two are identical, including details such as the size of memory and all the devices.

Deferring configuration until system startup allows the designer to build more robust code because a single system can operate on several hardware configurations. During startup, the system can adapt itself to the exact hardware on which it executes. In particular, dynamic configuration handles an arbitrary set of peripheral devices built by an arbitrary set of vendors. For example, a single operating system image can be used independent of the disk, printer, network interface, and video screen used on the computer. Dynamic reconfiguration also allows a system to adapt to changes in the hardware without stopping (e.g., when a user attaches or detaches a USB device).

25.3 Configuration In Xinu

Because it runs as an embedded system, Xinu follows a static configuration approach, with the majority of configuration occurring when the system is compiled and linked. Of course, even in some embedded systems, part of the configuration must be postponed until system startup. For example, some versions of Xinu calculate the size of memory during system initialization. Others use dynamic configuration to detect the presence of a real-time clock. As we have seen, some bus hardware chooses IRQs and device addresses when the bus is powered on. On such hardware, Xinu must wait until it runs to find the interrupt vector addresses and device CSR addresses.

To help manage configuration and to automate the selection of device driver modules, Xinu uses a separate configuration program. Named *config*, the program is not part of the operating system, and we do not need to examine the source code. Instead, we will look at how *config* operates: it takes an input file that contains specifications, and produces output files that become part of the operating system code. The next sections explain the configuration program and show examples.

25.4 Contents Of The Xinu Configuration File

The *config* program takes as input a text file named *Configuration*. It parses the input file, and generates two output files: *conf.h* and *conf.c*. We have already seen the output files, which contain defined constants for devices and a definition of the device switch table.†

The Xinu configuration file is a text file, divided into three sections. The sections are separated by lines that contain two ampersand characters (%%). The three sections are:

- Section 1: *Type declarations* for device types
- Section 2: *Device specifications* for specific devices
- Section 3: *Symbolic constants*

25.4.1 Section 1: Type Declarations

The motivation for a type declaration arises because a system may contain more than one copy of a particular hardware device. For example, a system may contain two UART devices that both use the tty abstraction. In such cases, a set of functions that comprise a tty driver must be specified for each UART. Entering the specification many times manually is error-prone, and can lead to inconsistencies. Thus, the type section allows the specification to be entered once and assigned a name that is used with both devices in the device specification section.

†File *conf.h* can be found on page 267, and *conf.c* can be found on page 279.

Each type declaration defines a name for the type, and lists a set of default device driver functions for the type. The declaration also allows one to specify the type of hardware with which the device is associated. For example, the type declaration:

```
tty:
        on uart
                -i ttyinit      -o ionull       -c ionull
                -r ttyread      -g ttygetc      -p ttyputc
                -w ttywrite     -s ioerr        -n ttycontrol
                -intr ttyhandler                -irq 8
```

defines a type named *tty* that is used on a UART device. Neither *tty* nor *uart* is a keyword or has any meaning. Instead they are simply names that the designer chose. The remaining items specify the default driver functions for type *tty*. Each driver function is preceded by a keyword that begins with a minus sign. Figure 25.1 lists the possible keywords and gives their meaning. Note: a given specification does not need to use all keywords. In particular, because a CSR address is unique to a device, a type declaration does not usually include a *csr* keyword.

Keyword	Meaning
-i	function that performs init
-o	function that performs open
-c	function that performs close
-r	function that performs read
-w	function that performs write
-s	function that performs seek
-g	function that performs getc
-p	function that performs putc
-n	function that performs control
-intr	function that handles interrupts
-csr	control and status register address
-irq	interrupt vector number

Figure 25.1 Keywords used in the Xinu configuration file and their meaning.

25.4.2 Section 2: Device Specifications

Section 2 of file *Configuration* contains a declaration for each device in the system. A declaration gives a name for the device (e.g., *CONSOLE*), and specifies the set of functions that constitutes a driver. Note that in Xinu, a device is an abstract concept, not necessarily tied to a physical hardware device. For example, in addition to devices like *CONSOLE* and *ETHERNET* that each correspond to underlying hardware, the device section can list *pseudo-devices*, such as a *FILE* device used for I/O.

Declaring a device serves two purposes. First, it allocates a slot in the device switch table, allowing the high-level I/O primitives to be used with the device rather than requiring a programmer to call specific driver functions. Second, it allows *config* to assign each device a minor device number. All devices with the same type are assigned minor numbers in sequence starting at zero.

When a device is declared, specific values can be supplied as needed or driver functions can be overridden. For example, the declaration:

```
CONSOLE is tty  on uart -csr 0xB8020000
```

declares *CONSOLE* to be a device of type *tty* that runs on UART hardware. In addition, the declaration specifies a CSR address of *0xB8020000*.

If a programmer needed to test a new version of *ttygetc*, the programmer might change the specification to:

```
CONSOLE is tty  on uart -csr 0xB8020000  -g myttygetc
```

which uses the default driver functions from the tty declaration given above, but overrides the *getc* function to use *myttygetc*. Note that having a configuration makes it easy to change one function without modifying or replacing the original file.

The example type declaration includes the phrase *on uart*. To understand the purpose of specifying the underlying hardware, observe that designers sometimes wish to use the same abstraction for multiple pieces of hardware. For example, suppose a system contains two types of UART hardware. The *on* keyword allows a designer to use the tty abstraction for hardware types, and to allocate a single array of control blocks even though some of the low-level hardware details differ and the set of driver functions used for one type of hardware differs from the set used for another.

25.4.3 Symbolic Constants

In addition to defining the structure of the device switch table, *conf.h* contains constants that specify the total number of devices and the number of each type. The *config* program generates such constants to reflect the specification found in file *Configuration*. For example, constant *NDEVS* is defined to be an integer that tells the total number of devices that have been configured. The device switch table contains *NDEVS* devices, and device-independent I/O routines use *NDEVS* to test whether a device id corresponds to a valid device.

Config also generates a set of defined constants that specify the number of devices of each type. Driver functions can use the appropriate constant to declare the array of control blocks. Each constant has the form N*xxx*, where *xxx* is the type name. For example, if file *Configuration* defines two devices of type *tty*, *conf.h* will contain the following line:

```
#define Ntty  2
```

25.5 Computation Of Minor Device Numbers

Consider the files that *config* produces. File *conf.h* contains the declaration of the device switch table, and *conf.c* contains the code that initializes the table. For a given device, its *devtab* entry contains a set of pointers to the device driver routines that correspond to high-level I/O operations like *open*, *close*, *read*, and *write*. The entry also contains the interrupt vector address and the device's CSR address (if the hardware allows the addresses to be known when the system is compiled). All information in the device switch table is derived from file *Configuration* in a straightforward manner.

As mentioned above, each entry in the device switch table also contains a *minor device number*. Minor device numbers are nothing more than integers that distinguish among multiple devices that each use the same type of control block. Recall that device driver functions use the minor device number as an index into the array of control blocks to associate a specific entry with each device. In essence, the *config* program counts devices of each type. That is, each time it encounters a device, config uses the device type to assign the next minor device number (numbers start at zero). For example, Figure 25.2 shows how device IDs and minor numbers are assigned on a system that has three *tty* devices and two *eth* devices.

device name	device identifier	device type	minor number
CONSOLE	0	tty	0
ETHERNET	1	eth	0
SERIAL2	2	tty	1
PRINTER	3	tty	2
ETHERNET2	4	eth	1

Figure 25.2 An example of device configuration.

Notice that the three *tty* lines have minor numbers zero, one, and two, even though their device IDs happen to be zero, two, and three.

25.6 Steps In Configuring A Xinu System

To configure a Xinu system, the programmer edits file *Configuration*, adding or changing device information and symbolic constants as desired. When run, program *config* first reads and parses the file, collecting the information about each device type. It then reads device specifications, assigns minor device numbers, and produces the output files *conf.c* and *conf.h*. Finally, in addition to symbolic constants that are generated

automatically, *config* appends definitions of symbolic constants from the third section of the specifications onto *conf.h*, making them available for operating system functions to include.

After *config* produces a new version of *conf.c* and *conf.h*, *conf.c* must be recompiled, as must all system functions that include *conf.h*.

25.7 Perspective

The history of operating systems is one of moving from static configuration to dynamic configuration. The interesting question is whether the benefits of dynamic configuration outweigh the costs. For example, compare the time required to boot Xinu to the time required to boot a large production system, such as Windows or Linux. Although the underlying computer hardware does not usually change, a production system may need to go through the steps of polling the bus to find the devices that are present, loading drivers, and interacting with each device

Some systems, especially those used on laptops, offer a *sleep mode* (sometimes called *hibernation mode*) that allows rapid restart. Instead of completely powering down the system, sleep mode saves processor state (e.g., by saving the running memory image and hardware registers on disk). Then, when a user restarts the machine, the machine state is restored from disk. Only a few checks must be made before the system can resume running. For example, if a laptop was connected to a wireless network before being put into sleep mode, the user could have moved to a new location where the network is no longer available. Therefore, the operating system must check network connections after the system resumes.

25.8 Summary

Instead of building a monolithic operating system tailored to specific hardware, designers look for ways to make systems configurable. Static configuration, a form of early binding, selects modules when the system is compiled and linked. The alternative, dynamic configuration, loads modules, such as device drivers, at runtime.

Because it is designed for an embedded environment, Xinu uses static configuration. Program *config* reads file *Configuration*, and produces files *conf.h* and *conf.c* that define and initialize the device switch table. The separation of device types from device declarations allows *config* to compute minor device numbers.

EXERCISES

25.1 Create a function *myttyread* that calls *ttygetc* repeatedly to satisfy a request. To test your code, modify file *Configuration* to substitute your code in place of *ttyread*.

25.2 Find out how other systems are configured. What happens, for example, when Windows boots?

25.3 If every operating system function includes *conf.h*, any change to file *Configuration* means a new version of *conf.h* will be generated, and the entire system must be recompiled. Redesign the *config* program to separate constants into several different include files to eliminate unnecessary recompilation.

25.4 Discuss whether a configuration program is worthwhile. Include some estimate of the extra effort required to make a system easily configurable. Remember that a programmer is likely to have little experience or knowledge about a system when it is first configured.

25.5 In theory, many aspects of a system may need to change when porting the system from one computer to another. In addition to devices, for example, one might consider the processor (not only the basic instruction set, but the extra instructions found on some models), the availability of co-processors (including floating point), the real-time clock or time resolution, and the endianness of integers. Argue that if a configuration system has parameters for all the above, the resulting system is untestable.

25.6 Measure the time a production system takes to boot and the time the same system takes to resume after hibernation. What is the percentage difference?

25.7 In the previous exercise, does the time to resume change if the system is moved and its network is no longer available?

25.8 Repeat the measurement of booting vs. resuming after hibernation for a system that has three USB devices plugged in before hibernation and none when the system resumes. Also repeat the experiment for the case where the system has no USB devices before hibernation, but three devices are plugged in during hibernation.

Chapter Contents

26.1 Introduction, 621
26.2 What Is A User Interface?, 622
26.3 Commands And Design Principles, 622
26.4 Design Decisions For A Simplified Shell, 623
26.5 Shell Organization And Operation, 623
26.6 The Definition Of Lexical Tokens, 624
26.7 The Definition Of Command-Line Syntax, 625
26.8 Implementation Of The Xinu Shell, 625
26.9 Storage Of Tokens, 628
26.10 Code For The Lexical Analyzer, 629
26.11 The Heart Of The Command Interpreter, 633
26.12 Command Name Lookup And Builtin Processing, 641
26.13 Arguments Passed To Commands, 641
26.14 Passing Arguments To A Non-builtin Command, 643
26.15 I/O Redirection, 646
26.16 An Example Command Function (sleep), 647
26.17 Perspective, 649
26.18 Summary, 650

26

An Example User Interface: The Xinu Shell

A man has to learn that he cannot command things...

— James Allen

26.1 Introduction

Previous chapters explain an operating system as a set of functions that applications can invoke to obtain services. However, a typical user never encounters system functions. Instead, users invoke applications, and allow the applications to access functions in the underlying system.

This chapter examines a basic interface known as a *shell*† that allows a user to launch applications and control their input and output. Following the pattern established by other parts of the system, our design emphasizes simplicity and elegance rather than features. We concentrate on a few fundamental ideas that make the shell powerful without requiring large amounts of code. The chapter shows examples of both software that interprets user commands and applications that a user can invoke. Although it only offers basic functionality, our example interpreter illustrates several important concepts.

†The term *shell* and many of the ideas used in the Xinu shell come from Unix.

26.2 What Is A User Interface?

A *user interface* consists of the hardware and software with which users interact to perform computational tasks and observe the results. Thus, user interface software lies between a human who specifies what must be done and a computer system that performs the specified tasks.

The goal of user interface design is to create an environment in which users can perform computational tasks conveniently and productively. For example, most modern user interfaces employ a graphical representation that presents a set of icons from which the user selects to launch an application. The use of graphics makes application selection quick, and relieves the user from memorizing a set of application names.

Small embedded systems typically offer two levels of interface: one for a system builder and another for the end user. For example, a desktop system provides a graphical interface through which an end user launches applications and interacts with the device. To write new software, directly access the file system, or enter text commands, however, a programmer invokes a lower-level interface that is sometimes called a *Terminal*.

26.3 Commands And Design Principles

Industry uses the term *Command Line Interface* (*CLI*) to describe an interface that allows a user to enter a series of textual commands; many embedded system products offer a CLI. Usually, each line of input corresponds to a command, and the system processes a line before reading the next line. The term *command* arises because most CLIs follow the same syntactic form in which a line starts with a name that specifies an action to be taken and is followed by parameters that control the details of the action and the item(s) to which the action applies. For example, we can imagine a system that uses a command named *config* to control settings associated with a network interface. To minimize keystrokes, a CLI may use short, cryptic abbreviations. The command to set the MTU parameter of interface number 0 to 1500 might be:

config 0 MTU=1500

The set of all available commands determines the functionality available to a user (i.e., defines the power of the computing system). However, a good design does not merely collect random commands. Instead, the design adheres to the following principles:

- Functionality: sufficient for all needs
- Orthogonality: only one way to perform a given task
- Consistency: commands follow a consistent pattern
- Least astonishment: a user should be able to predict results

26.4 Design Decisions For A Simplified Shell

When they design a CLI and the shell program that processes commands, programmers must choose among many alternatives. The following paragraphs list decisions a programmer faces, and describe the choices made for a simplified Xinu shell.

Handling input. Should our interface allow the terminal device driver to handle the details of backspacing, character echoing, and line erasing, or should it handle those details itself? The choice is important because it determines the extent to which the shell can control input. For example, a modern Unix shell allows *Control-B* and *Control-F* characters to move the cursor backward and forward while editing a line of input, while our tty driver does not.† Our choice is to use the tty driver, which simplifies the design and reduces the amount of code. An exercise suggests changing the design to have the shell, rather than the tty driver, interpret all input characters.

Foreground or background execution. Does the shell wait while a command completes execution before starting another? Our shell follows the Unix tradition of allowing a user to decide whether the shell waits or the command executes in foreground or background.

Redirection of input and output. Also like Unix, our example shell allows a user to specify a source of input and destination for output when the command is invoked. The technique, known as *I/O redirection*, allows each command to function as a general-purpose tool that can be applied to a variety of files and I/O devices. Providing redirection in the shell also means that I/O specifications are uniform — a single mechanism for redirection applies to all commands.

Typed or untyped arguments. Should a shell check the number and type of arguments for a given command, or should each command handle arguments? Following the Unix tradition, our example shell does not check the number of arguments, nor does it interpret argument values. Instead, the shell treats each argument as a text string, and the set of arguments is passed to the command. Consequently, each command must check whether its arguments are valid.

26.5 Shell Organization And Operation

A shell is organized as a loop that repeatedly reads a line of input and executes the command on the line. Once a line has been read, the shell must extract a command name, arguments, and other items, such as the specifications of I/O redirection or background processing. Following standard practice for syntactic analysis, we have divided the code into two separate functions: one function handles lexical analysis by grouping characters into *tokens*, and the other function examines whether the set of tokens forms a valid command.

Using a separate lexical function may seem unnecessary for the trivial syntax of our sample shell. It is tempting to take short cuts for a small task. However, we chose the organization because it permits future expansion.

---

†The use of *Control-B* and *Control-F* is derived from the Emacs editor.

26.6 The Definition Of Lexical Tokens

At the lexical level, our shell scans a line of input and groups characters into syntactic tokens. Figure 26.1 lists the lexical tokens that the scanner recognizes and the four lexical types the scanner uses when classifying tokens.

Symbolic Name (Token Type)	Numeric Value	Input Characters	Description
SH_TOK_AMPER	0	&	ampersand
SH_TOK_LESS	1	<	less-than symbol
SH_TOK_GREATER	2	>	greater-than symbol
SH_TOK_OTHER	3	'...'	quoted string (single quotes)
SH_TOK_OTHER	3	"..."	quoted string (double-quotes)
SH_TOK_OTHER	3	other	sequence of non-whitespace

Figure 26.1 Lexical tokens used by the example Xinu shell.

As with most command line processors, our shell strives to be flexible. Characters such as a "less than" (i.e., <) are assigned special meaning. The shell includes quoted strings as lexical tokens to permit a user to include special characters in arguments. The rule is that a quoted string can contain arbitrary characters, including the special characters recognized by the shell. Rather than incorporate complex rules for escaping characters in a string, the Xinu shell uses two forms. A quoted string starts with either a single or double quote, and contains all characters, including blanks, tabs, and special characters up to the first occurrence of the opening quote. Thus, the string:

```
'a string'
```

contains eight total characters including a blank. More important, because two quotes are recognized, a string can be created that includes one of the quote characters. For example, the string:

```
"don't blink"
```

contains eleven characters, including a single quote. The lexical scanner removes the surrounding quotes, and classifies the resulting sequence of characters to be a single token of type *SH_TOK_OTHER*.

The lexical scanner defines *whitespace* to consist of blanks or tab characters. At least one whitespace character must separate two tokens of type *SH_TOK_OTHER*. Otherwise, whitespace is ignored.

26.7 The Definition Of Command-Line Syntax

Once a line has been scanned and divided into a series of lexical tokens, the shell parses the tokens to verify that they form a valid sequence. The syntax is:

command_name args* [redirection] [background]

Brackets [ ] denote an optional occurrence and asterisk indicates zero or more repetitions of an item. The string *command_name* denotes the name of a command, *args** denotes zero or more optional arguments, the optional *redirection* refers to input redirection, output redirection, or both, and the optional *background* indicates background execution. Figure 26.2 contains a grammar that defines the set of valid inputs in terms of tokens.

command	→	**name [args] [redirection] [background]**
name	→	**SH_TOK_OTHER**
args	→	**SH_TOK_OTHER [args]**
redirection	→	**input_redirect [output_redirect]**
redirection	→	**output_redirect [input_redirect]**
input_redirect	→	**SH_TOK_LESS SH_TOK_OTHER**
output_redirect	→	**SH_TOK_GREATER SH_TOK_OTHER**
background	→	**SH_TOK_AMPER**

Figure 26.2 A grammar that specifies valid sequences of tokens for the example shell.

In essence, a command consists of a sequence of one or more "other" tokens optionally followed by requests to redirect the input and/or output (in either order), optionally followed by a specification that the command should run in background. The first token on the line must be the name of a command.

26.8 Implementation Of The Xinu Shell

Our examination of the implementation begins with the definition of constants and variables used by the shell. File *shell.h* contains the declarations.

```
/* shell.h - Declarations and constants used by the Xinu shell */

/* Size constants */

#define SHELL_BUFLEN    TY_IBUFLEN+1    /* Length of input buffer       */
#define SHELL_MAXTOK    32              /* Maximum tokens per line      */
#define SHELL_CMDSTK    8192            /* Size of stack for process    */
                                        /*    that executes command     */
#define SHELL_ARGLEN    (SHELL_BUFLEN+SHELL_MAXTOK) /* Argument area    */
#define SHELL_CMDPRIO   20              /* Process priority for command */

/* Message constants */

/* Shell banner (assumes VT100) */

#define SHELL_BAN0      "\033[1;31m"
#define SHELL_BAN1      "-----------------------------------------"
#define SHELL_BAN2      "   __    __   _____   _   _   _   _    "
#define SHELL_BAN3      "   \\ \\  / /  |__ __|  | \\ | |  | |  | |   "
#define SHELL_BAN4      "    \\ \\/ /     | |    |  \\| |  | |  | |   "
#define SHELL_BAN5      "    / /\\ \\    _| |_   | \\   |  | |  | |   "
#define SHELL_BAN6      "   / /  \\ \\  |     |  | | \\ |  \\  --  /   "
#define SHELL_BAN7      "   --    --   -----    -   -     ----     "
#define SHELL_BAN8      "-----------------------------------------"
#define SHELL_BAN9      "\033[0;39m\n"

/* Messages shell displays for user */

#define SHELL_PROMPT    "xsh $ "        /* Command prompt               */
#define SHELL_STRTMSG   "Welcome to Xinu!\n"/* Welcome message          */
#define SHELL_EXITMSG   "Shell closed\n"/* Shell exit message           */
#define SHELL_SYNERRMSG "Syntax error\n"/* Syntax error message         */
#define SHELL_CREATMSG  "Cannot create process\n"/* command error      */
#define SHELL_INERRMSG  "Cannot open file %s for input\n" /* Input err  */
#define SHELL_OUTERRMSG "Cannot open file %s for output\n"/* Output err */
                                        /* Builtin cmd error message    */
#define SHELL_BGERRMSG  "Cannot redirect I/O or background a builtin\n"

/* Constants used for lexical analysis */

#define SH_NEWLINE      '\n'            /* New line character           */
#define SH_EOF          '\04'           /* Control-D is EOF             */
#define SH_AMPER        '&'             /* Ampersand character          */
#define SH_BLANK        ' '             /* Blank character              */
#define SH_TAB          '\t'            /* Tab character                */
```

```
#define SH_SQUOTE       '\''            /* Single quote character       */
#define SH_DQUOTE       '"'             /* Double quote character       */
#define SH_LESS         '<'             /* Less-than character  */
#define SH_GREATER      '>'             /* Greater-than character       */

/* Token types */

#define SH_TOK_AMPER    0               /* Ampersand token              */
#define SH_TOK_LESS     1               /* Less-than token              */
#define SH_TOK_GREATER  2               /* Greater-than token           */
#define SH_TOK_OTHER    3               /* Token other than those       */
                                        /*   listed above (e.g., an     */
                                        /*   alphanumeric string)       */

/* Shell return constants */

#define SHELL_OK         0
#define SHELL_ERROR      1
#define SHELL_EXIT      -3

/* Structure of an entry in the table of shell commands */

struct  cmdent  {                       /* Entry in command table       */
        char    *cname;                 /* Name of command              */
        bool8   cbuiltin;               /* Is this a builtin command?   */
        int32   (*cfunc)(int32,char*[]);/* Function for command         */
};

extern  uint32  ncmd;
extern  const   struct  cmdent  cmdtab[];
```

Unlike a Unix system, where commands reside in files, all Xinu shell commands are linked into the image. The shell defines an array that lists the set of commands that are available, the name of each command, and the function to execute for the command.

The final section of file *shell.h* defines the table of commands, *cmdtab*. Each entry in the table is struct *cmdent* which contains three items: a name for the command, a Boolean indicating whether the command is restricted to run as a builtin, and a pointer to a function that implements the command. Later sections discuss how the command table is initialized and how it is used.

26.9 Storage Of Tokens

The data structures that our shell uses are somewhat unexpected. An integer array, *toktyp*, is used to record the numeric value of the type for each token. The tokens themselves are stored as null-terminated strings. To conserve space, tokens are packed into contiguous locations of a character array, *tokbuf*. An integer array, *tok*, is used to store the index of the start of each token. The shell depends on two counters: *ntok* counts the number of tokens found so far, and variable *tlen* counts the characters that have been stored in array *tokbuf*. To understand the data structures, consider the example input line:

date > file &

The line contains four tokens. Figure 26.3 shows how the lexical analyzer fills in the data structures to hold the tokens that have been extracted from the input line.

Figure 26.3 Contents of variables *tokbuf*, *toktyp*, *tok*, *ntok*, and *tlen* for the input line: *date* > *file* &

As the figure shows, the tokens themselves are placed in array *tokbuf* with all whitespace removed. A null character terminates each token. Array *tok* contains integers that are each an index into *tokbuf* — the ith location of array *tok* gives the index in *tokbuf* of the string for the ith token. Finally, the ith location of array *toktyp* specifies the type of the ith token. For example, the second token on the line, **>**, has numeric type 2 (*SH_TOK_LESS*†), and starts at location 5 in array *tokbuf*. The third token on the line, *file*, has numeric type 3 (*SH_TOK_OTHER*), and starts at location 7 in array *tokbuf*.

†Figure 26.1 on page 624 lists the numeric values for each token type.

26.10 Code For The Lexical Analyzer

Because our shell syntax is straightforward, we have chosen to use an ad hoc, top-down implementation for the lexical analyzer. File *lexan.c* contains the code.

```
/* lexan.c  - lexan */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lexan  -  Ad hoc lexical analyzer to divide command line into tokens
 *------------------------------------------------------------------------
 */

int32   lexan (
          char          *line,          /* Input line terminated with   */
                                        /*   NEWLINE or NULLCH          */
          int32         len,            /* Length of the input line,    */
                                        /*   including NEWLINE          */
          char          *tokbuf,        /* Buffer into which tokens are */
                                        /*   stored with a null         */
                                        /*   following each token       */
          int32         *tlen,          /* Place to store number of     */
                                        /*   chars in tokbuf            */
          int32         tok[],          /* Array of pointers to the     */
                                        /*   start of each token        */
          int32         toktyp[]        /* Array that gives the type    */
                                        /*   of each token              */
        )
{
        char    quote;                  /* Character for quoted string  */
        uint32  ntok;                   /* Number of tokens found       */
        char    *p;                     /* Pointer that walks along the */
                                        /*   input line                 */
        int32   tbindex;                /* Index into tokbuf            */
        char    ch;                     /* Next char from input line    */

        /* Start at the beginning of the line with no tokens */

        ntok = 0;
        p = line;
        tbindex = 0;

        /* While not yet at end of line, get next token */
```

```
while ( (*p != NULLCH) && (*p != SH_NEWLINE) ) {

        /* If too many tokens, return error */

        if (ntok >= SHELL_MAXTOK) {
                return SYSERR;
        }

        /* Skip whitespace before token */

        while ( (*p == SH_BLANK) || (*p == SH_TAB) ) {
                p++;
        }

        /* Stop parsing at end of line (or end of string) */

        ch = *p;
        if ( (ch==SH_NEWLINE) || (ch==NULLCH) ) {
                *tlen = tbindex;
                return ntok;
        }

        /* Set next entry in tok array to be an index to the    */
        /*   current location in the token buffer               */

        tok[ntok] = tbindex;    /* the start of the token       */

        /* Set the token type */

        switch (ch) {

            case SH_AMPER:      toktyp[ntok] = SH_TOK_AMPER;
                                tokbuf[tbindex++] = ch;
                                tokbuf[tbindex++] = NULLCH;
                                ntok++;
                                p++;
                                continue;

            case SH_LESS:       toktyp[ntok] = SH_TOK_LESS;
                                tokbuf[tbindex++] = ch;
                                tokbuf[tbindex++] = NULLCH;
                                ntok++;
                                p++;
                                continue;
```

```
    case SH_GREATER:    toktyp[ntok] = SH_TOK_GREATER;
                        tokbuf[tbindex++] = ch;
                        tokbuf[tbindex++] = NULLCH;
                        ntok++;
                        p++;
                        continue;

    default:            toktyp[ntok] = SH_TOK_OTHER;
};

/* Handle quoted string (single or double quote) */

if ( (ch==SH_SQUOTE) || (ch==SH_DQUOTE) ) {
        quote = ch;     /* remember opening quote */

        /* Copy quoted string to arg area */

        p++;    /* Move past starting quote */

        while ( ((ch=*p++) != quote) && (ch != SH_NEWLINE)
                        && (ch != NULLCH) ) {
                tokbuf[tbindex++] = ch;
        }
        if (ch != quote) {  /* string missing end quote */
                return SYSERR;
        }

        /* Finished string - count token and go on      */

        tokbuf[tbindex++] = NULLCH; /* terminate token  */
        ntok++;         /* count string as one token    */
        continue;       /* go to next token             */
}

/* Handle a token other than a quoted string            */

tokbuf[tbindex++] = ch; /* put first character in buffer*/
p++;

while ( ((ch = *p) != SH_NEWLINE) && (ch != NULLCH)
        && (ch != SH_LESS)  && (ch != SH_GREATER)
        && (ch != SH_BLANK) && (ch != SH_TAB)
        && (ch != SH_AMPER) && (ch != SH_SQUOTE)
        && (ch != SH_DQUOTE) )  {
```

```
                        tokbuf[tbindex++] = ch;
                        p++;
                }

                /* Report error if other token is appended */

                if (       (ch == SH_SQUOTE) || (ch == SH_DQUOTE)
                        || (ch == SH_LESS)   || (ch == SH_GREATER) ) {
                        return SYSERR;
                }

                tokbuf[tbindex++] = NULLCH;     /* terminate the token  */

                ntok++;                         /* count valid token    */

        }
        *tlen = tbindex;
        return ntok;
}
```

The first two arguments of *lexan* give the address of an input line and the number of characters on the line. Succeeding arguments give pointers to the data structures that Figure 26.3 illustrates. *Lexan* initializes the number of tokens found, a pointer to the input line, and an index into array tokbuf. It then enters a *while* loop that runs until pointer *p* reaches the end of the line.

To process a token, *lexan* skips leading whitespace (i.e., blanks and tabs), stores the current index of *tokbuf* in array *tok*, and then uses a switch statement to choose the action appropriate for the next input character. For any of the single-character tokens (i.e., an ampersand, less-than symbol, or greater-than symbol), *lexan* records the token type in array *toktyp*, places the token followed by a null character in the *tokbuf* array, increments *ntok*, moves to the next character in the string, and continues the while loop, which will start to process the next input character.

For a character that is not one of the three single-character tokens, *lexan* records the token type as *SH_TOK_OTHER* and exits the switch statement. There are two cases: the token is a quoted string or the token consists of contiguous characters up to the next special character or whitespace. *Lexan* recognizes either a single-quote or a double-quote character; the string ends at the first occurrence of a matching quote or the end-of-line, whichever occurs first. If it encounters an end-of-line condition in a string, *lexan* returns *SYSERR*. Otherwise, it copies characters from the string to *tokbuf* unchanged and uninterpreted, which means that a string can contain arbitrary characters, including whitespace and the other quote mark character. Once the copy has been completed, *lexan* appends a null character to define the end of the token. It then continues the outer while loop to look for the next token.

The final section of code handles a token that is composed of contiguous characters other than the single token characters and quotes. The code loops until it encounters a special character or whitespace, placing characters in successive locations of *tokbuf*. Before moving on to the next token, the code checks for an error where two tokens occur with no whitespace between them.

Once *lexan* reaches the end of the input line, it returns a count of the number of tokens found. If an error is detected during processing, *lexan* returns *SYSERR* to its caller, making no attempt to recover or repair the problem. That is, the action to be taken when an error occurs is coded into *lexan*. The exercises discuss the choice of error handling and suggest alternatives.

26.11 The Heart Of The Command Interpreter

Although a command interpreter must handle many details, the basic algorithm is not difficult to understand. At the top level, the code consists of a loop that repeatedly reads a line of input, uses *lexan* to extract tokens, checks the syntax, arranges a way to pass arguments, redirects I/O if necessary, and runs a command in foreground or background as specified. The loop terminates if a user enters the end-of-file character (control-d), or if a command returns a special exit code.

As with the lexical analyzer, our interpreter implementation uses an ad hoc implementation. That is, the code does not resemble a conventional compiler, nor does it contain independent code to verify that the sequence of tokens is valid. Instead, error checking is built into each step of processing. For example, after it has processed background and I/O redirection, the shell verifies that remaining tokens are all of type *SH_TOK_OTHER*.

Examining the code will make the approach clear. Function *shell* performs command interpretation; file *shell.c*, shown below, contains the code. Note that the file also includes the declaration of array *cmdtab* which specifies the set of commands and the function used to implement each. In addition, the code sets external variable *ncmd* to the number of commands in the table.

Conceptually, the set of commands is independent from the code that processes user input. Thus, it may make sense to divide *shell.c* into two files: one that specifies commands and another that contains code. In practice, however, the two have been combined because the example set of commands is so small that an additional file is unnecessary.

```
/* shell.c  -  shell */

#include <xinu.h>
#include <stdio.h>
#include "shprototypes.h"

/************************************************************************/
/* Table of Xinu shell commands and the function associated with each   */
/************************************************************************/
const   struct  cmdent  cmdtab[] = {
        {"argecho",     TRUE,   xsh_argecho},
        {"arp",         FALSE,  xsh_arp},
        {"cat",         FALSE,  xsh_cat},
        {"clear",       TRUE,   xsh_clear},
        {"date",        FALSE,  xsh_date},
        {"devdump",     FALSE,  xsh_devdump},
        {"echo",        FALSE,  xsh_echo},
        {"exit",        TRUE,   xsh_exit},
        {"help",        FALSE,  xsh_help},
        {"ipaddr",      FALSE,  xsh_ipaddr},
        {"kill",        TRUE,   xsh_kill},
        {"memdump",     FALSE,  xsh_memdump},
        {"memstat",     FALSE,  xsh_memstat},
        {"ping",        FALSE,  xsh_ping},
        {"ps",          FALSE,  xsh_ps},
        {"sleep",       FALSE,  xsh_sleep},
        {"udp",         FALSE,  xsh_udpdump},
        {"udpecho",     FALSE,  xsh_udpecho},
        {"udpeserver",  FALSE,  xsh_udpeserver},
        {"uptime",      FALSE,  xsh_uptime},
        {"?",           FALSE,  xsh_help}

};

uint32  ncmd = sizeof(cmdtab) / sizeof(struct cmdent);

/************************************************************************/
/* shell  -  Provide an interactive user interface that executes        */
/*           commands.  Each command begins with a command name, has    */
/*           a set of optional arguments, has optional input or         */
/*           output redirection, and an optional specification for      */
/*           background execution (ampersand).  The syntax is:          */
/*                                                                      */
/*                 command_name [args*] [redirection] [&]               */
/*                                                                      */
```

```
/*          Redirection is either or both of:                          */
/*                                                                     */
/*                              < input_file                           */
/*                      or                                             */
/*                              > output_file                          */
/*                                                                     */
/***********************************************************************/

process shell (
                did32   dev             /* ID of tty device from which  */
        )                               /*   to accept commands         */
{
        char    buf[SHELL_BUFLEN];      /* Input line (large enough for */
                                        /*   one line from a tty device */
        int32   len;                    /* Length of line read          */
        char    tokbuf[SHELL_BUFLEN +   /* Buffer to hold a set of      */
                        SHELL_MAXTOK];  /* Contiguous null-terminated   */
                                        /* Strings of tokens            */
        int32   tlen;                   /* Current length of all data   */
                                        /*   in array tokbuf            */
        int32   tok[SHELL_MAXTOK];      /* Index of each token in       */
                                        /*   array tokbuf               */
        int32   toktyp[SHELL_MAXTOK];   /* Type of each token in tokbuf */
        int32   ntok;                   /* Number of tokens on line     */
        pid32   child;                  /* Process ID of spawned child  */
        bool8   backgnd;                /* Run command in background?   */
        char    *outname, *inname;      /* Pointers to strings for file */
                                        /*   names that follow > and <  */
        did32   stdinput, stdoutput;    /* Descriptors for redirected   */
                                        /*   input and output           */
        int32   i;                      /* Index into array of tokens   */
        int32   j;                      /* Index into array of commands */
        int32   msg;                    /* Message from receive() for   */
                                        /*   child termination          */
        int32   tmparg;                 /* Address of this var is used  */
                                        /*   when first creating child  */
                                        /*   process, but is replaced   */
        char    *src, *cmp;             /* Pointers used during name    */
                                        /*   comparison                 */
        bool8   diff;                   /* Was difference found during  */
                                        /*   comparison                 */
        char    *args[SHELL_MAXTOK];    /* Argument vector passed to    */
                                        /*   builtin commands           */

        /* Print shell banner and startup message */
```

```
fprintf(dev, "\n\n%s%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n",
        SHELL_BAN0,SHELL_BAN1,SHELL_BAN2,SHELL_BAN3,SHELL_BAN4,
        SHELL_BAN5,SHELL_BAN6,SHELL_BAN7,SHELL_BAN8,SHELL_BAN9);

fprintf(dev, "%s\n\n", SHELL_STRTMSG);

/* Continually prompt the user, read input, and execute command */

while (TRUE) {

        /* Display prompt */

        fprintf(dev, SHELL_PROMPT);

        /* Read a command */

        len = read(dev, buf, sizeof(buf));

        /* Exit gracefully on end-of-file */

        if (len == EOF) {
                break;
        }

        /* If line contains only NEWLINE, go to next line */

        if (len <= 1) {
                continue;
        }

        buf[len] = SH_NEWLINE;  /* terminate line */

        /* Parse input line and divide into tokens */

        ntok = lexan(buf, len, tokbuf, &tlen, tok, toktyp);

        /* Handle parsing error */

        if (ntok == SYSERR) {
                fprintf(dev,"%s\n", SHELL_SYNERRMSG);
                continue;
        }

        /* If line is empty, go to next input line */

        if (ntok == 0) {
```

```
        fprintf(dev, "\n");
        continue;
}

/* If last token is '&', set background */

if (toktyp[ntok-1] == SH_TOK_AMPER) {
        ntok-- ;
        tlen-= 2;
        backgnd = TRUE;
} else {
        backgnd = FALSE;
}


/* Check for input/output redirection (default is none) */

outname = inname = NULL;
if ( (ntok >=3) && ( (toktyp[ntok-2] == SH_TOK_LESS)
                    ||(toktyp[ntok-2] == SH_TOK_GREATER))){
        if (toktyp[ntok-1] != SH_TOK_OTHER) {
                fprintf(dev,"%s\n", SHELL_SYNERRMSG);
                continue;
        }
        if (toktyp[ntok-2] == SH_TOK_LESS) {
                inname =  &tokbuf[tok[ntok-1]];
        } else {
                outname = &tokbuf[tok[ntok-1]];
        }
        ntok -= 2;
        tlen = tok[ntok];
}


if ( (ntok >=3) && ( (toktyp[ntok-2] == SH_TOK_LESS)
                    ||(toktyp[ntok-2] == SH_TOK_GREATER))){
        if (toktyp[ntok-1] != SH_TOK_OTHER) {
                fprintf(dev,"%s\n", SHELL_SYNERRMSG);
                continue;
        }
        if (toktyp[ntok-2] == SH_TOK_LESS) {
                if (inname != NULL) {
                    fprintf(dev,"%s\n", SHELL_SYNERRMSG);
                    continue;
                }
```

```
                inname = &tokbuf[tok[ntok-1]];
        } else {
                if (outname != NULL) {
                    fprintf(dev,"%s\n", SHELL_SYNERRMSG);
                    continue;
                }
                outname = &tokbuf[tok[ntok-1]];
        }
        ntok -= 2;
        tlen = tok[ntok];
}

/* Verify remaining tokens are type "other" */

for (i=0; i<ntok; i++) {
        if (toktyp[i] != SH_TOK_OTHER) {
                break;
        }
}
if ((ntok == 0) || (i < ntok)) {
        fprintf(dev, SHELL_SYNERRMSG);
        continue;
}

stdinput = stdoutput = dev;

/* Lookup first token in the command table */

for (j = 0; j < ncmd; j++) {
        src = cmdtab[j].cname;
        cmp = tokbuf;
        diff = FALSE;
        while (*src != NULLCH) {
                if (*cmp != *src) {
                        diff = TRUE;
                        break;
                }
                src++;
                cmp++;
        }
        if (diff || (*cmp != NULLCH)) {
                continue;
        } else {
                break;
        }
```

```
        }

        /* Handle command not found */

        if (j >= ncmd) {
                fprintf(dev, "command %s not found\n", tokbuf);
                continue;
        }

        /* Handle built-in command */

        if (cmdtab[j].cbuiltin) { /* No background or redirect. */
                if (inname != NULL || outname != NULL || backgnd){
                        fprintf(dev, SHELL_BGERRMSG);
                        continue;
                } else {
                        /* Set up arg vector for call */

                        for (i=0; i<ntok; i++) {
                                args[i] = &tokbuf[tok[i]];
                        }

                        /* Call builtin shell function */

                        if ((*cmdtab[j].cfunc)(ntok, args)
                                                == SHELL_EXIT) {
                                break;
                        }
                }
                continue;
        }

        /* Open files and redirect I/O if specified */

        if (inname != NULL) {
                stdinput = open(NAMESPACE,inname,"ro");
                if (stdinput == SYSERR) {
                        fprintf(dev, SHELL_INERRMSG, inname);
                        continue;
                }
        }
        if (outname != NULL) {
                stdoutput = open(NAMESPACE,outname,"w");
                if (stdoutput == SYSERR) {
                        fprintf(dev, SHELL_OUTERRMSG, outname);
```

```
                        continue;
                } else {
                        control(stdoutput, F_CTL_TRUNC, 0, 0);
                }
        }

        /* Spawn child thread for non-built-in commands */

        child = create(cmdtab[j].cfunc,
                SHELL_CMDSTK, SHELL_CMDPRIO,
                cmdtab[j].cname, 2, ntok, &tmparg);

        /* If creation or argument copy fails, report error */

        if ((child == SYSERR) ||
            (addargs(child, ntok, tok, tlen, tokbuf, &tmparg)
                                                 == SYSERR) ) {
                fprintf(dev, SHELL_CREATMSG);
                continue;
        }

        /* Set stdinput and stdoutput in child to redirect I/O */

        proctab[child].prdesc[0] = stdinput;
        proctab[child].prdesc[1] = stdoutput;

        msg = recvclr();
        resume(child);
        if (! backgnd) {
                msg = receive();
                while (msg != child) {
                        msg = receive();
                }
        }
    }

    /* Terminate the shell process by returning from the top level */

    fprintf(dev,SHELL_EXITMSG);
    return OK;
}
```

The main loop calls *lexan* to divide the input line into tokens, and begins processing the command. First, the code checks the last token to see if the user appended an

ampersand. If so, the shell sets Boolean *backgnd* to *TRUE*; otherwise, *backgnd* is set to *FALSE*. The variable is used later to determine whether to run the command in background.

After the background token has been removed, the shell checks for I/O redirection. Input and output redirection can both be specified, and the specifications can occur in either order, but must be the last of the remaining tokens. Therefore, the shell checks for redirection twice. If two specifications occur, the shell verifies that they do not both specify input or both specify output. At this point in processing the line, the shell merely saves a pointer to the file name without attempting to open the file (the files are opened later).

Once the shell has removed tokens that specify I/O redirection, the only tokens that remain correspond to a command name and arguments to the command. Thus, before it continues to process the command, the shell iterates through remaining tokens to verify that they are of type "other" (*SH_TOK_OTHER*). If any are not, the code prints an error message and moves to the next input line. Once all checks have been performed, the shell looks up the command and executes the corresponding function.

26.12 Command Name Lookup And Builtin Processing

The first token on the line is taken to be a command name. Recall that the example code stores information about commands in array *cmdtab*. Thus, lookup is straightforward — the shell searches the array sequentially looking for an exact match between the first token and one of the command names. If no match is found, the code prints an error message and moves to the next command.

Our shell supports two types of commands: builtin and non-builtin. The difference arises from the way the commands are executed: the shell uses the conventional function call mechanism to execute a builtin command, and creates a separate process to execute a non-builtin command. The distinction means that a user cannot specify background processing and cannot redirect I/O for a builtin command.†

To test whether a command should execute as a builtin, the shell examines field *cbuiltin* of the entry in *cmdtab*. For a builtin command, no redirection or background processing is allowed. So, the shell verifies that neither was specified, creates an argument array in variable *args*, and calls the command function. The next section explains how command arguments are constructed.

26.13 Arguments Passed To Commands

Our example shell uses the same argument passing mechanism as a Unix shell. When a command is invoked, the shell passes tokens from the command line as uninterpreted, null-terminated strings. The shell does not know how many arguments a given command expects, nor does the shell understand whether the arguments make sense.

†An exercise suggests a way to blur the distinction between builtin and non-builtin commands.

Instead, the shell merely passes all arguments from the command line, and allows the command to check and interpret them.

Conceptually, the shell passes an arbitrary number of string arguments, where the number is only limited by the length of an input line. To make programming simple and uniform, the shell creates an array of pointers and only passes two values when it invokes a command: a count of command-line arguments and an array of pointers to character strings that constitute the arguments. Unix uses the names *argc* and *argv* for the two arguments a command receives from the shell; Xinu uses the names *nargs* and *args*. The names are only a convention — a programmer can choose arbitrary names for arguments when writing a function that implements a command.

The example shell adopts another convention from Unix: the first item in the *args* array is a pointer to the command name. An example will clarify the details. Consider the command line:

```
date -f illegal
```

Although the argument *illegal* is not permitted by the Xinu *date* command, the shell simply passes the string and allows the function that implements the *date* command to check its arguments. Figure 26.4 illustrates the two items the shell passes to the date function.

Figure 26.4 Illustration of the two arguments (*nargs* and *args*) the shell passes to the date command for an input line: *date -f illegal*

Although passing an integer, such as *nargs*, to a command is trivial, the *args* array is more complex. In essence, the shell must construct the *args* array, and then pass its address to the command. There are two cases: builtin commands and non-builtin commands. We will consider builtins first.

After the shell has parsed the command line and removed tokens for I/O redirection and background processing, variable *ntok* will contain the count of remaining tokens, which is exactly the count needed for *nargs*. Furthermore, array *tok* contains the index in *tokbuf* where each token begins. Therefore, the shell can create an *args* array by computing the starting address of each token.

To form the *args* array, the code iterates through *ntok* tokens and for the ith token computes the expression:

```
&tokbuf[tok[i]]
```

That is, it sets *args[i]* equal to the address of the ith token in *tokbuf*. Once array *args* has been initialized, the shell calls the function that implements the builtin command.

26.14 Passing Arguments To A Non-builtin Command

The second case, non-builtin commands, is more complex. Our shell creates a separate process to execute commands that are not builtin, and the command can execute in background (i.e., the shell can continue to read and handle new input lines while the command process runs in background). The question arises: what mechanism should the shell use to pass arguments to a process? The shell cannot use the same approach as with builtin commands because a command running in background needs a separate copy of its arguments that will not change as the shell goes on to process another command.

There are two ways to solve the problem of argument passing for a non-builtin command: the shell can allocate separate storage to arguments or the shell can hide the arguments in storage already allocated for the process. Because Xinu does not automatically release heap storage when a process terminates, the first approach requires the shell to keep a record of the argument storage allocated for each command so it can free the memory once the process completes. Thus, we have chosen the second approach:

> *After creating a process to execute a command, the shell places a copy of arguments in the stack area of the process and then allows the process to execute.*

Where on the process's stack should arguments be placed? Although it might be possible to rewrite *create* so it leaves space at the top of the stack, doing so is messy. Thus, we have chosen to use the area at the bottom of the stack. The shell stores a copy of the *args* array followed by a copy of the strings in *tokbuf* in the stack. Of course, pointers in the copy of the *args* vector must be assigned the addresses of strings in the copy of *tokbuf*. Figure 26.5 illustrates how the data from Figure 26.4 is arranged in contiguous memory locations.

Figure 26.5 A copy of the *args* array and argument strings in a process's stack.

Rather than incorporate code into the shell that copies items into the process's stack, our implementation uses a separate function, *addargs*. File *addargs.c* contains the code.

```
/* addargs.c - addargs */

#include <xinu.h>
#include "shprototypes.h"

/*------------------------------------------------------------------------
 *  addargs  -  Add local copy of argv-style arguments to the stack of
 *                a command process that has been created by the shell
 *------------------------------------------------------------------------
 */
status  addargs(
          pid32         pid,            /* ID of process to use         */
          int32         ntok,           /* Count of arguments           */
          int32         tok[],          /* Index of tokens in tokbuf    */
          int32         tlen,           /* Length of data in tokbuf     */
          char          *tokbuf,        /* Array of null-term. tokens   */
          void          *dummy          /* Dummy argument that was      */
                                        /*   used at creation and must  */
                                        /*   be replaced by a pointer   */
                                        /*   to an argument vector      */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  procent *prptr;         /* Ptr to process' table entry  */
        uint32  aloc;                   /* Argument location in process */
                                        /*   stack as an integer        */
        uint32  *argloc;                /* Location in process's stack  */
                                        /*   to place args vector       */
        char    *argstr;                /* Location in process's stack  */
                                        /*   to place arg strings       */
        uint32  *search;                /* pointer that searches for    */
                                        /*   dummy argument on stack    */
        uint32  *aptr;                  /* Walks through args array     */
        int32   i;                      /* Index into tok array         */

        mask = disable();

        /* Check argument count and data length */

        if ( (ntok <= 0) || (tlen < 0) ) {
```

```
        restore(mask);
        return SYSERR;
}

prptr = &proctab[pid];

/* Compute lowest location in the process stack where the       */
/*      args array will be stored followed by the argument      */
/*      strings                                                 */

aloc = (uint32) (prptr->prstkbase
        - prptr->prstklen + sizeof(uint32));
argloc = (uint32*) ((aloc + 3) & ~0x3); /* round multiple of 4  */

/* Compute the first location beyond args array for the strings */

argstr = (char *) (argloc + (ntok+1));  /* +1 for a null ptr    */

/* Set each location in the args vector to be the address of    */
/*      string area plus the offset of this argument            */

for (aptr=argloc, i=0; i < ntok; i++) {
        *aptr++ = (uint32) (argstr + tok[i]);
}

/* Add a null pointer to the args array */

*aptr++ = (uint32)NULL;

/* Copy the argument strings from tokbuf into process's stack   */
/*      just beyond the args vector                             */

memcpy(aptr, tokbuf, tlen);

/* Find the second argument in process's stack */

for (search = (uint32 *)prptr->prstkptr;
     search < (uint32 *)prptr->prstkbase; search++) {

        /* If found, replace with the address of the args vector*/

        if (*search == (uint32)dummy) {
                *search = (uint32)argloc;
                restore(mask);
                return OK;
        }
}
```

```
        /* Argument value not found on the stack - report an error */

        restore(mask);
        return SYSERR;
}
```

Once a process has been created, the process table entry contains both the address of the top of the stack and the stack size. Because a stack grows downward in memory, *addargs* can compute the lowest memory address assigned to the stack by subtracting the stack size from the address of the stack top. However, a few details complicate the code. For example, on some processors pointers must be aligned. Therefore, *addargs* computes a starting location in the stack that is a multiple of four bytes. As a result, the final byte of the last argument string may end up to three bytes before the lowest byte of the stack. Furthermore, the code adds an extra null pointer to the end of the *args* array as shown in Figure 26.5.

Most of code in *addargs* operates as expected by computing the address in the stack at which the *args* array starts and then copying both the *args* array and the argument strings into the stack. However, the final *for* loop, which iterates through the process's stack may seem unusual: it finds the second argument that has been passed to the process and replaces it with a pointer to the *args* array. When the process is created, the shell uses a dummy value for the argument and then passes the value to *addargs* in parameter *dummy*. Thus, *addargs* searches the stack until it finds the value and replaces it.

Why does our implementation use a dummy argument and a search? The alternative consists of having *addargs* calculate the location of the second argument. Although calculating a location may seem cleaner, such a calculation requires *addargs* to understand the format of the initial process stack. Using a search means that only *create* needs to understand the details of process creation and the format of the stack. Of course, using a search has a disadvantage: the shell must choose a dummy argument that will not occur early in the stack. Rather than choosing an arbitrary integer, our shell implementation uses the address of variable *tmparg*.

26.15 I/O Redirection

Once a process has been created to execute a command and the shell has called *addargs* to copy arguments into the process stack, all that remains is to handle I/O redirection and start the process executing. To redirect I/O, the shell assigns device descriptors to the array *prdesc* in the process table entry. The two key values are *prdesc[0]* and *prdesc[1]*, which the shell sets to *stdin* and *stdout*.

How do variables *stdin* and *stdout* receive values? Recall that the shell initializes them to *dev*, the device descriptor that was passed as an argument when the shell was invoked. Usually, the shell is invoked with device *CONSOLE*. Thus, if the user does not redirect I/O, the process executing a command will "inherit" the console device for

input and output. If a user does redirect I/O, the shell sets variables *inname* and/or *outname* to the name that was specified on the command line. Otherwise *inname* and *outname* are set to NULL. Before assigning *stdin* and *stdout* to the command process, the shell checks *inname* and *outname*. If *inname* is non-null, the shell calls *open* to open *inname* for reading and sets *stdin* to the descriptor. Similarly, if *outname* is non-null, the shell calls *open* to open *outname* for writing, and sets *stdout* to the descriptor.

When should descriptors be closed? Our example code assumes that the command will close standard input and standard output descriptors before it exits; the shell does not clean up descriptors after the command completes. Forcing all commands to close their standard I/O devices before exiting has the disadvantage of making commands difficult to understand and difficult to program correctly because command code must remember to close devices even though the code does not open them. Having the shell monitor command processes and close standard I/O devices is also difficult because command processes are independent and multiple command processes can exit at the same time. The exercises suggest another alternative.

The final section of code in the shell runs the command process. There are two cases. To run the process in foreground, the shell calls *resume* to start the process, and then calls *receive* to wait for a message that the process has completed (when the process exits, *kill* sends a message to the shell). For the background case, the shell starts the command process but does not wait. Instead, the main shell loop continues and the shell reads the next command. The exercises suggest a modification of the code to improve correctness.

26.16 An Example Command Function (sleep)

To understand how a command processes arguments, consider function *xsh_sleep*, which implements the *sleep* command.† *Sleep* is trivial — it delays for the number of seconds specified by its argument. Thus, the delay is achieved by a single line of code that calls the *sleep* system function; the code is presented here merely to illustrate how arguments are parsed and how a command function prints a help message. File *xsh.sleep.c* contains the code.

†By convention, the function that implements command *X* is named *xsh_X*.

```
/* xsh_sleep.c - xsh_sleep */

#include <xinu.h>
#include <stdio.h>
#include <string.h>

/*------------------------------------------------------------------------
 * xsh_sleep  -  Shell command to delay for a specified number of seconds
 *------------------------------------------------------------------------
 */
shellcmd xsh_sleep(int nargs, char *args[])
{
        int32   delay;                  /* Delay in seconds             */
        char    *chptr;                 /* Walks through argument       */
        char    ch;                     /* Next character of argument   */

        /* For argument '--help', emit help about the 'sleep' command   */

        if (nargs == 2 && strncmp(args[1], "--help", 7) == 0) {
                printf("Use: %s\n\n", args[0]);
                printf("Description:\n");
                printf("\tDelay for a specified number of seconds\n");
                printf("Options:\n");
                printf("\t--help\t display this help and exit\n");
                return 0;
        }

        /* Check for valid number of arguments */

        if (nargs > 2) {
                fprintf(stderr, "%s: too many arguments\n", args[0]);
                fprintf(stderr, "Try '%s --help' for more information\n",
                                args[0]);
                return 1;
        }

        if (nargs != 2) {
                fprintf(stderr, "%s: argument in error\n", args[0]);
                fprintf(stderr, "Try '%s --help' for more information\n",
                                args[0]);
                return 1;
        }

        chptr = args[1];
        ch = *chptr++;
```

```
        delay = 0;
        while (ch != NULLCH) {
                if ( (ch < '0') || (ch > '9') ) {
                        fprintf(stderr, "%s: nondigit in argument\n",
                                args[0]);
                        return 1;
                }
                delay = 10*delay + (ch - '0');
                ch = *chptr++;
        }
        sleep(delay);
        return 0;
}
```

26.17 Perspective

The design of a shell introduces many choices. A designer has almost complete freedom because a shell operates as an application that lies outside the rest of the system and only the command functions depend on the shell. Thus, as our example shows, the argument passing paradigm used by a shell can differ dramatically from the argument passing paradigm used throughout the rest of the system. Similarly, a designer can choose a syntax for command-line input as well as a semantic interpretation without affecting other parts of the system.

Perhaps the most interesting aspect of shell design arises from the choice of how much knowledge about commands is bound into the shell. On the one hand, if a shell knows all commands and their arguments, the shell can complete command names and check the command arguments, making the code that implements commands much simpler. On the other hand, allowing late binding means more flexibility because the shell does not need to change when new commands are created, but the tradeoff is that each command must check its arguments. Furthermore, a designer can choose whether to build each command function into the shell or to leave each command in a separate file, as Unix does.

Our example shell demonstrates one of the most important principles in shell design: a relatively small amount of code can provide powerful abstractions for a user. For example, consider how little code is needed to interpret input or output redirection, and the small amount of code needed to recognize an ampersand at the end of the line as a request to run a command in background. Despite their compact implementation, facilities for I/O redirection and background processing make a shell much more powerful and user-friendly than a shell in which each command interacts with a user to prompt for input and output information or whether to run in background.

26.18 Summary

We have examined a basic command-line interpreter called a *shell*. Although the example code is small, it supports concurrent command execution, redirection of input and output, and arbitrary string argument passing. The implementation is divided into two conceptual pieces: a lexical analyzer that reads a line of text and groups characters into tokens, and a shell function that checks the sequence of tokens and invokes a command.

The example code demonstrates the relationship between a user interface and the facilities provided by the underlying system. For example, although the underlying system provides support for concurrent processes, the shell makes concurrent execution available to a user. Similarly, although the underlying system provides the ability to open devices or files, the shell makes I/O redirection available to the user.

EXERCISES

26.1 Rewrite the shell to use *raw* mode and handle all keyboard input. Arrange for *Control-P* to move to the "previous" shell command, and interpret *Control-B* and *Control-F* as moving backward and forward through a line as Unix *ksh* or *bash* do.

26.2 Rewrite the grammar in Figure 26.2 to remove the optional notation [ ].

26.3 Modify the shell to allow I/O redirection on builtin commands. What changes are necessary?

26.4 Devise a modified version of *create* that handles string arguments for the shell, automatically performing the same function as *addargs* when creating the process.

26.5 Modify the shell so it can be used as a command. That is, allow the user to invoke command *shell* and start with a new shell process. Have control return to the original shell when the subshell exits. Be careful.

26.6 Modify the shell so it can accept input from a text file (i.e., allow the user to build a file of commands and then start a shell interpreting them).

26.7 Modify the shell to allow a user to redirect standard error as well as standard output.

26.8 Read about shell variables in a Unix shell, and implement a similar variable mechanism in the Xinu shell.

26.9 Find out how the Unix shell passes environment variables to command processes, and implement a similar mechanism in the Xinu shell.

26.10 Implement inline input redirection, allowing the user to type

```
command << stop
```

followed by lines of input terminated by a line that begins with the sequence of characters *stop*. Have the shell save the input in a temporary file and execute the command with the temporary file as standard input.

26.11 It is possible to extend the command table to include information on the number and types of arguments each command requires, and to have the shell check arguments before passing them to the command. List two advantages and two disadvantages of having the shell check arguments.

26.12 Suppose the designer decided to add a *for* statement to the shell so a user could execute a command repeatedly as in:

```
for 1 2 3 4 5 6 7 8 9; command-line
```

where *for* is a keyword, and *command-line* is a command line exactly like commands the shell now accepts. Should the designer modify the shell syntax and parser or try to make *for* a builtin command? Explain.

26.13 Add command completion to the shell by having the user type a unique prefix of a command followed by *ESC*; have the shell type out the completed command and wait for the user to add arguments and press the *Enter* key.

26.14 Unix allows command lines of the form:

```
command_1 | command_2 | command_3
```

where the symbol |, called a *pipe*, specifies that the standard output of one command is connected to the standard input of the next command. Implement a *pipe* device for Xinu, and modify the shell to allow a pipeline of commands.

26.15 Modify the *tty* device driver and shell so that pressing CONTROL-c kills the currently executing process.

26.16 Modify the *tty* device driver and shell so that pressing CONTROL-z places the currently executing process in background.

26.17 Modify the design to permit I/O redirection and background processing for builtin commands: if redirection or background processing is needed, treat the command as a normal command and create a separate process.

26.18 The text describes the problem of closing device descriptors when a command exits. Modify the system so *kill* automatically closes a process's descriptors when the process exits.

26.19 The example shell calls *receive* to wait for a foreground process to exit, but does not check the message received. Show a sequence of events that can cause the shell to proceed before a foreground command has completed.

26.20 Modify the code in the shell to repair the problem in the previous exercise.

Appendix 1

Porting An Operating System

Progress, far from consisting in change, depends upon retentiveness ....

— George Santayana

A1.1 Introduction

Throughout the text, we have focused on the interior of an operating system. Chapters present abstractions, discuss design tradeoffs, show how the code fits into a hierarchical organization, and examine implementation details. Chapter 25 examines how a system can be configured to allow the code to run on systems with a variety of peripheral devices.

This appendix examines two larger questions. First, how can an existing operating system be ported to a new hardware platform that differs in some fundamental ways? Second, can an operating system be written in a way that makes porting easier? To answer the first question, the appendix discusses cross-development and downloading, outlines the steps involved in understanding hardware, and provides practical advice about how to proceed. To answer the second question, the appendix discusses techniques that have been used to make an operating system adaptable.

A1.2 Motivation: Evolving Hardware

Although work on operating systems demands the ability to grasp high-level abstractions, design efficient mechanisms, and understand small details, the most significant challenge facing operating system designers does not arise from the intellectual difficulty of the task. Instead, it arises from the constant changes in technology and the consequent economic pressure for vendors to create new products or to add features to existing products. While this revision of the text was being written, for example, one hardware vendor introduced two new hardware platforms. One of the platforms represented a minor improvement, but the other made dramatic changes in the processor chip, instruction set, memory organization, and I/O devices.

Because an operating system interfaces directly with the underlying hardware, even small changes in the hardware can have an overwhelming effect on the system. For example, if a vendor changes the hardware to reserve a piece of the memory address space for Flash ROM, the memory management software in the operating system must be modified. Although such modification may seem straightforward, the details may involve changes to page tables and the code that interacts with the MMU hardware as well as the code that allocates memory on demand. If a significant amount of the address space becomes reserved, the operating system's allocation policy may need to change. The point is:

> *Because both technological and economic factors cause continual changes in hardware, an operating system designer must be prepared to port the operating system to new platforms.*

A1.3 Steps Taken When Porting An Operating System

Despite the effects of hardware change, moving an existing operating system to a new platform is much easier than designing and building a new system from scratch. In particular, if an operating system has been written in a high-level language, porting the system to a new platform is easy because a compiler can do most of the work.

Consider Xinu. Most of the code is written in C. If a C compiler is available for the new platform, many of the functions can be compiled without making changes in the source. If a given function deals with basic data structures, such as integers, characters, arrays, and structures, a compiler may be able to compile the code without change, and the resulting binary program may run correctly. Even in cases where change is needed, the modifications may be minor (e.g., to accommodate slight differences in compilers). Thus, a principle is:

> *An operating system written in a high-level language, such as C, is much easier to port to a new platform than a system written in assembly language.*

We will assume that, whenever possible, an operating system has been written in C, and we will consider the steps taken when porting to a new platform. Specifically, Figure A1.1 lists steps taken when porting Xinu.

Step	Description
1.	Learn about the new hardware
2.	Build cross-development tools
3.	Learn the compiler's calling conventions
4.	Build a bootstrap mechanism
5.	Devise a basic polled output function
6.	Load and run a sequential program
7.	Port and test the basic memory manager
8.	Port the context switch and process creation functions
9.	Port and test the remaining process manager functions
10.	Build an interrupt dispatcher
11.	Port and test the real-time clock functions
12.	Port and test a tty driver
13.	Port or create drivers for an Ethernet and other devices
14.	Port the network stack, including Internet protocols
15.	Port the remote disk and RAM disk modules
16.	Port the local and remote file system modules
17.	Port the Xinu shell and other applications

Figure A1.1 The steps taken when porting Xinu to a new platform.

Note the relationship between the steps listed and the Xinu hierarchy. In essence, porting follows the same pattern as design: lower levels of the hierarchy are ported first, and successive levels are then added. The next sections highlight each step.

A1.3.1 Learning About The Hardware

Step 1 may seem straightforward. Unfortunately, some vendors are reluctant to reveal details about their commercial hardware or software. For example, one vendor of a popular platform for experimenters refuses to provide any details about the USB hardware, making it impossible to write a USB driver. Even if generic information is available (e.g., the processor instruction set), a vendor may choose to keep some details secret (e.g., a map of the bus address space, the hardware initialization sequence, or the details about addresses on the bus). Nevertheless, throughout the remainder of this appendix, we will assume that the needed information can be obtained.

A1.3.2 Build Cross-Development Tools

If the target hardware platform already runs a fully functional operating system, it may be possible to skip steps 1 through 6, and use the legacy operating system to compile and boot a new system. In most cases, however, the target platform will be new, may lack the power needed for a production system, or may not be available for development. Therefore, an operating system designer usually does not rely on the target platform to support software development. Instead, a designer uses a *cross-development* approach in which a compiler and linker run on a conventional computer system, but are configured to produce code for the target machine.

One of the most widely used cross-development environments consists of the *Gnu C Compiler*, *gcc*, running on a Unix system, such as Linux or BSD. A copy of *gcc* can be downloaded and used at no cost from:

http://gcc.gnu.org

After downloading the source code for *gcc*, a programmer must select configuration options to specify details, such as the target processor type and the endianness of the target machine. The programmer runs the Unix utility *make* to build a version of the compiler, assembler, and linker that will produce code for the target machine.

A1.3.3 Learn The Compiler's Calling Conventions

Because it is a key to basic parts of an operating system, function invocation must be understood in detail. To build a context switch, for example, a programmer must precisely understand all details of the calling conventions. Although hardware designers include subroutine invocation mechanisms, understanding the hardware is not enough because a compiler can impose additional requirements.

It may seem that using an open source compiler means the calling conventions will be obvious. However, an operating system designer needs to know about special cases, and the answers to questions may be difficult to find. Fortunately, information is usually available on the web.

A1.3.4 Build A Bootstrap Mechanism

Once it has been compiled and linked, a program image must be downloaded into the target platform. Early embedded hardware required that the image be burned into a separate ROM chip and installed in a socket. Fortunately, modern systems use alternative mechanisms that require much less effort. Typically, the hardware includes bootstrap functionality that can read an image from an SD memory card or a USB device, accept an image over a console serial line, or download an image over a network. The bootstrap procedure is usually intended for system developers, and may not generally be known.

For example, consider the platforms used in the text. One can get started by building an image, storing the image on an SD memory card, and booting from the card. Manually moving a card from a development machine to the target platform after every change can be tedious. Downloading over a network means a developer can create a new image and boot it without touching the hardware. However, arranging a way to download an image over a network can sometimes be difficult. For example, the Galileo does not have facilities that can boot an image over a network, which is why we had to write a bootstrap program, place the program on an SD card, and arrange for the program to perform a network download. Whatever method is chosen, it is essential to find a way to boot a copy of an image on the target machine.

A1.3.5 Devise A Basic Polled Output Function

The next step in porting requires a programmer to devise a way for a running program to output characters. Until some basic I/O is available, a programmer must work in the dark — hoping that the image has been downloaded and started correctly. Some I/O is invaluable: once even basic I/O is available, a programmer can determine how much of the system is working, and can isolate problems quickly.

Because early test programs do not include interrupt processing, the basic I/O mechanism must use *polling*. A typical approach starts by finding a way to light an LED (e.g., by using GPIO pins). The key to development, however, lies in a *kputc* function that waits for a serial I/O device to become ready and then transmits a character. Both ends must agree on details such as the baud rate and bits per character, which can make debugging tedious. To simplify the code, the first version of *kputc* can be written in assembly language and can have information about the device (e.g., the CSR address and the baud rate) hardwired into the program. Once character output is available, debugging proceeds rapidly.

A1.3.6 Load And Run A Sequential Program

Once an image can be downloaded and run, the next step consists of building an environment that permits a sequential program to execute. In particular, successful execution of a C program requires that memory permissions are set correctly (the program text is readable and data locations can be read and written) and a runtime stack exists (which is needed for function calls).

Initializing the environment may seem trivial, but it requires knowledge of both hardware and compiler details. For example, one must choose a mode for the processor. One must also choose an initial value for a stack pointer that does not conflict with device CSRs or holes in memory. Depending on the platform and compiler, it may be advisable to initialize a frame pointer or exception vectors (e.g., to prevent an invalid memory reference from generating another invalid memory reference, which generates another invalid memory reference, and so on).

A1.3.7 Port And Test The Basic Memory Manager

Once the layout of memory is known and a sequential program can be downloaded and run on the target hardware, a programmer can port and test the four basic memory management functions: *getmem*, *freemem*, *getstk*, and *freestk*. In addition to basic allocation and deallocation tests, a programmer should concentrate on alignment. Some hardware platforms require all memory accesses to be word aligned and some allow unaligned access. On machines that require alignment, a programmer should ensure that the memory free list has been initialized in such a way that alignment works correctly (i.e., all allocated blocks begin on the appropriate boundary).

A1.3.8 Port The Context Switch And Process Creation Functions

Once basic memory management is working, a programmer can begin to port the process manager functions. In particular, a programmer starts with context switch, scheduling, and process creation. A giant step forward occurs once the three fundamental process management functions are in place: instead of a sequential program, the code will be a fledgling operating system that supports concurrent execution.

There are two difficult parts in the design. Creating the saved information for a process requires an intricate knowledge of the machine state and the operation of the context switch. Building a context switch is tricky because it involves finding a way to save all the state associated with a process and reload all the state from another process. It can be easy to overlook details or inadvertently to destroy state while saving a copy (e.g., clobber a register). Unfortunately, debugging can be extremely difficult because problems may not be discovered until the system attempts to reload saved state.

A1.3.9 Port And Test The Remaining Process Manager Functions

Once process creation, scheduling, and context switching are working, other process management functions can be added easily. Semaphores functions can be ported and tested, as can message passing. Beyond the context switch level, most process management functions do not depend on the hardware. Of course, various data types may change, depending on the underlying hardware. For example, when moving from a 32-bit computer to a 64-bit computer, the *msg32* type may be changed to *msg64*. Nevertheless, porting the semaphore and message passing functions is a relatively straightforward task.

A1.3.10 Build An Interrupt Dispatcher

The last big hardware hurdle concerns interrupts. Building an interrupt dispatcher requires a detailed knowledge of the hardware. How do the processor, interrupt controller, and bus interact? Exactly what state does the hardware save when an interrupt occurs, and what state is the operating system required to save? How does the dispatcher determine which device interrupted? How does a dispatcher return to the running program when the interrupt ends? What addresses are used for the bus and devices?

The details of interrupts are surprisingly subtle. On most systems, for example, I/O is memory mapped. Thus, I/O devices (and perhaps the bus hardware) are mapped into specific addresses. To access I/O facilities, however, an operating system may need to disable or avoid the memory cache because each I/O access must go to the underlying hardware rather than the cache.

A1.3.11 Port And Test The Real-Time Clock Functions

Once interrupt dispatching is in place, an example device is needed to test the mechanism. It is logical to start by testing the real-time clock, usually one of the easiest devices to use. On some systems building a real-time clock handler first is absolutely necessary because the clock cannot be stopped — if the system has interrupts enabled, clock interrupts will occur. Clock interrupts mean that processes can call *sleep()* to delay for a specified time, and that time slicing goes into effect.

A1.3.12 Port And Test A Tty Driver

Clock interrupts are distinct from other devices because a clock does not perform input or output. A serial line is perhaps the simplest type of device that has both input and output (some hardware separates input and output interrupt handling). Thus, the tty driver will exercise both input and output, and ensure that all basic interrupt processing works.

Fortunately, most systems include a serial line, and many use the same UART hardware as described in the text. Thus, much of the tty driver code, including the lower half, can simply be recompiled and used. Basic device parameters, which will have been worked out in Step A1.3.5, can be added to the device switch table or the lower half as appropriate.

A1.3.13 Port Or Create Drivers For An Ethernet And Other Devices

Once input and output have been tested, more complex device drivers can be ported. Devices that use DMA (e.g., disks and network interfaces) require buffer pools to be in place, and may require a deeper understanding of how DMA interacts with a memory cache. However, having a basic system in place makes debugging much easier because a programmer can focus on one device at a time.

A1.3.14 Port The Network Stack, Including Internet Protocols

Because networking has become fundamental, network protocols are an essential part of an operating system. Some embedded systems that use network communication do not have a local disk. Once an Ethernet driver is available, adding higher layer protocols is straightforward. Porting can start with a network input process; UDP, IP, and ARP can then be added. Once UDP is available, Internet communication can be tested.

A1.3.15 Port The Remote Disk And RAM Disk Modules

Porting a RAM disk driver is trivial, and allows a programmer to test the local file system. The remote disk driver provides access to stable storage, even if the platform does not have a disk. Thus, porting the RAM disk and remote disk is an easy step before porting a file system.

A1.3.16 Port The Local And Remote File System Modules

Given an operational disk driver, porting a basic file system is straightforward. The first step consists of porting and testing functions that read and write index blocks; the second step consists of porting and testing code that builds the free lists of index and data blocks. Once the basic allocation functions are in place, a directory can be added and the file system tested.

A1.3.17 Port The Xinu Shell And Other Applications

Although applications are convenient, a shell adds significant complexity to the system. Thus, one usually writes functions that test each operating system module and exercise special cases. Once the system is running, the final step consists of porting a shell and more general-purpose applications.

A1.4 Programming To Accommodate Change

How should operating system designers contend with constant change? Can we anticipate future hardware? Can a system be designed and implemented to make changes easier? Designers have been considering the questions for decades. Most early operating systems were created to match the hardware and written in assembly language. Each system was designed and built from scratch, with new abstractions and new mechanisms. As I/O devices (such as disks) and operating system abstractions (such as files) became standardized, designing a new system from scratch became much more expensive than adapting an existing system. Modern systems employ two techniques to accommodate change:

- Compile time: write source code that can generate multiple versions.
- Runtime: design facilities that allow an operating system to change dynamically.

Compile time. One way to make a system adaptable consists of writing source code that uses conditional compilation to allow a given source program to be used on multiple systems. As a simplistic example, consider writing an operating system that must run on hardware with a real-time clock or hardware that has no real-time clock. A programmer can use the C preprocessor to conditionally compile source code according to the hardware. For example, if preprocessor variable *RT_CLOCK* has been defined, functions that use the clock should be compiled as usual. Otherwise, functions that depend on a clock should be replaced by versions that report an error. The *sleep* function from Chapter 13 can be used to illustrate the idea. To accommodate both situations, the code from the function can be rewritten as follows:

```
syscall sleep(
          uint32        delay           /* Time to delay in seconds     */
        )
{
#ifdef RT_CLOCK
        if (delay > MAXSECONDS) {
                return(SYSERR);
        }
        sleepms(1000*delay);
        return OK;
#else
        return SYSERR;
#endif
}
```

If constant *RT_CLOCK* has been defined, the C preprocessor generates the source code shown in Chapter 13, which is then compiled. If *RT_CLOCK* has not been defined, the C preprocessor eliminates the body of the *sleep* function and generates a single line of source code:

```
return SYSERR;
```

The chief advantage of conditional compilation lies in its efficiency: instead of using a test at runtime, the source code can be tailored to the specific hardware. Furthermore, the system does not contain extra code that is never used (which can be important in embedded systems). The chief disadvantage of conditional compilation is loss of readability. In the case above, the code snippet is so small that the entire block of code fits onto a display. In a production system, the conditional block may span hundreds of lines of code. Furthermore, if two conditions interact, conditional code may be nested.

Runtime. The simplest way to increase runtime portability consists of using conditional execution. When it starts, the operating system gathers information about the hardware and places the information in a global data structure. For example, a Boolean variable in the global data structure might specify whether the hardware includes a real-time clock. Each operating system function is written to interrogate the data structure and act accordingly. The chief advantage of using a runtime approach lies in generality — an image can be run without being recompiled.

The idea of runtime adaptation has been generalized by separating an operating system into two parts: a *microkernel* that contains basic process management functionality and a series of dynamically loaded kernel modules that extend the functionality. In theory, porting a microkernel to a new environment is easier than porting a complete system because porting can be done piecemeal. That is, the microkernel is ported first, and modules are ported later, as needed.

A1.5 Summary

Portability is important because hardware continues to change. The steps required to port an operating system to a new environment follow the same pattern as the original design: port the lower levels of the system first, and then port successively higher levels.

Operating system code can be written to increase portability. A compile-time approach that uses conditional compilation achieves highest efficiency. A runtime approach that uses conditional execution allows a single image to run on multiple versions of a platform. The most advanced runtime approach uses a microkernel plus dynamic kernel modules, which allows modules to be ported only if they are needed.

Appendix 2

Xinu Design Notes

A2.1 Introduction

This appendix contains a set of informal notes that characterize Xinu and the underlying design. The notes are not intended as a tutorial, nor are they a complete description of the system. Instead, they provide a concise summary of characteristics and features.

A2.2 Overview

Embedded paradigm. Because it is intended for use in embedded systems, Xinu follows a cross-development paradigm. To develop create, edit, cross-compile, and cross-link Xinu software, a programmer uses a conventional computer, typically one that runs a version of the Unix operating system, such as Linux. Output from cross-development software is a memory image. Once such an image has been created, a programmer downloads the image to the target system (typically over a computer network). Finally, the programmer starts the image running on the target embedded system.

Source code organization. Xinu software is organized into a handful of directories that follow the organization used with various Unix systems. Instead of placing all files for each module in a separate directory, files are grouped into a few directories. For example, all include files are placed in one directory, and files that constitute the kernel sources are placed in another. Device drivers are the exception — source files for a given device driver are placed in a subdirectory named for the device type. The subdirectories within a Xinu source tree are organized as follows:

./compile	**The Makefile used to compile and link a Xinu image**
/bin	**Executable scripts invoked during compilation**
/binaries	**Compiled binaries for Xinu functions (.o files)**
./config	**Source for the configuration program and Makefile**
/conf.h	**Configuration include file (copied to ../include)**
/conf.c	**Configuration declarations (copied to ../system)**
./device	**Source code for device drivers, organized into one subdirectory for each device type**
/tty	**Source code for the tty driver**
/rfs	**Source code for the remote file access system (both the master device and remote file pseudo-devices**
/eth	**Source code for the Ethernet driver**
/rds	**Source code for the remote disk driver**
/…	**Directories for other device drivers**
./include	**All include files**
./shell	**Source code for the Xinu shell and shell commands**
./system	**Source code for Xinu kernel functions**
./lib	**Source code for library functions**
./net	**Source code for network protocol software**

A2.3 Xinu Characteristics

Note: these are the notes kept during implementation; they are not intended to be a tutorial introduction to Xinu.

- The system supports multiple concurrent processes.
- Each process is known by its process ID.
- The process ID is used as an index into the process table.
- The system provides counting semaphores.
- Each semaphore is known by its ID, which is used as an index into the semaphore table.

- The system supports a real-time clock that is used for round-robin scheduling of equal-priority processes and timed delays.
- Each process is assigned a priority, which is used in scheduling; a process priority can be changed dynamically.
- The system supports multiple I/O devices and multiple types of I/O devices.
- The system includes a set of device-independent I/O primitives.
- The console device uses a tty abstraction in which characters are queued both during input and output.
- The tty driver supports modes: raw mode passes characters transparently, and cooked mode supports character echo, line editing with erasing backspace, flow control, and *crlf* mapping.
- The system includes an Ethernet driver that can send and receive Ethernet packets; the driver uses DMA.
- The system includes a local file system that supports concurrent growth of files without preallocation; the local file system only has a single-level directory structure.
- Xinu also includes a remote file system mechanism that allows access to files on a remote server.
- The system includes a message passing mechanism used for interprocess communication; each message is one word long.
- Processes are dynamic — a process can be created, suspended, restarted, and killed.
- Xinu includes a low-level memory manager used to allocate and free heap areas or process stacks, and a high-level memory manager used to create buffer pools, where each pool contains a set of fixed-size buffers.
- Xinu includes a configuration program that generates a Xinu system according to the specifications given; the configuration program allows one to choose a set of devices and set system parameters.
- The system provides Internet access via TCP and UDP.

A2.4 Xinu Implementation

Functions and modules. The system sources are organized as a set of functions. In general, each file corresponds to a system call (e.g., file *resume.c* contains system call *resume*). In addition to the system call function, a file may contain utility functions needed by that system call.

Key files. In most cases, each major system function is found in a separate file. For example, function *resume* is found in file *resume.c*. The following lists other files that play an important role in Xinu.

Configuration	A text file containing device information and constants that describe the system and the hardware. The *config* program takes file *Configuration* as input and produces *conf.c* and *conf.h*.
conf.h	Generated by *config*, it contains declarations and constants including defined names of I/O devices, such as *CONSOLE*.
conf.c	Generated by *config*, it contains initialization for the device switch table.
kernel.h	General symbolic constants and type declarations used throughout the kernel.
prototypes.h	Prototype declarations for all system functions.
xinu.h	A master include file that includes all header files in the correct order. Most Xinu functions only need to include *xinu.h*.
process.h	Process table entry structure declaration; state constants.
semaphore.h	Semaphore table entry structure declaration; semaphore constants.
tty.h	Tty device control block, buffers, and other tty constants.
bufpool.h	Buffer pool constants and format.
memory.h	Constants and structures used by the low-level memory manager.
ports.h	Definitions by the high-level inter-process communication mechanism.
sleep.h	Definitions for real-time delay functions.
queue.h	Declarations and constants for the general-purpose process queue manipulation functions.
resched.c	The Xinu scheduler that selects the next process to run from the eligible set; *resched* calls the context switch.
ctxsw.S	The context switch that changes from one executing process to another; it consists of a small piece of assembly code.
initialize.c	The system initialization function, *sysinit*, and other initialization code as well as code for the null process (process 0).

userret.c The function to which a user process returns if the process exits. *Userret* must never return. It must kill the process that executed it because the stack does not contain a legal frame or return address.

platinit.c Platform-specific initialization.

A2.5 Major Concepts And Implementation

Process states. Each process has a state given by field *prstate* in its process table entry. Constants that define process states have names of the form *PR_xxxx*. *PR_FREE* means the process entry is unused. *PR_READY* means the process is linked into the ready list and is eligible for the processor. *PR_WAIT* means the process is waiting on a semaphore (given by prsem). *PR_SUSP* means the process is in hibernation; it is not on any list. *PR_SLEEP* means the process is in the queue of sleeping processes and will awaken after a timeout. *PR_CURR* means that the process is (the only one) currently running. The currently running process is not on the ready list. *PR_RECV* means the process is blocked waiting to receive a message; *PR_RECTIM* is like *PR_RECV* except the process is also sleeping for a specified time and will awaken if the timer expires or a message arrives, whichever happens first.

Counting semaphores. Semaphores reside in the array *semtab*. Each entry in the array corresponds to a semaphore and has a count (*scount*) and state (*sstate*). The state is *S_FREE* if the semaphore slot is unassigned, and *S_USED* if the semaphore is in use. If the count is negative *P* then the head and tail fields of the entry in the semaphore table point to the head and tail of a FIFO queue of *P* processes waiting for the semaphore. If the count is nonnegative *P* then no processes are waiting and the queue is empty.

Blocked processes. A process that is blocked for any reason is not eligible to use the processor. Any action that blocks the current processes forces it to relinquish the processor and allow another process to execute. A process that is blocked on a semaphore is on the queue for the semaphore, and a process blocked for a timed delay is on the queue of sleeping processes. Other blocked processes are not on a queue. Function *ready* moves a blocked process to the ready list and makes the process eligible to use the processor.

Sleeping processes. A process calls *sleep* to delay for a specified time. The process is added to a delta list of sleeping processes. A process may only put itself to sleep.

Process queues and ordered lists. There is a single data structure used for all process lists. The structure contains entries for the head and tail of each list as well as an entry for each process. The first *NPROC* entries in the table (0 to *NPROC-1*) correspond to the *NPROC* processes in the system; successive entries in the table are allocated in pairs, where each pair forms the head and tail of a list.

The advantage of keeping all heads and tails in the same data structure is that enqueuing, dequeuing, testing for empty/nonempty, and removing from the middle (e.g., when a process

is killed) are all handled by a small set of functions (files *queue.c* and *queue.h*). An empty queue has the head and tail pointing to each other. Testing whether a list is empty is trivial. Lists can be ordered or may be FIFO; each entry has a key that is ignored if the list is FIFO.

Null process. Process 0 is a null process that is always available to run or is running. Care must be taken so that process 0 never executes code that could cause it to block (e.g., it cannot wait for a semaphore). Because the null process may be running during interrupts, interrupt code may never wait for a semaphore. When the system starts, the initialization code creates a process to execute *main* and then becomes the null process (i.e., executes an infinite loop). Because its priority is lower than that of any other process, the null process loop executes only when no other process is ready.

Netin process. The network input process, *netin* repeatedly reads and demultiplexes incoming packets. *Netin* must not block or all network input stops. Therefore, if an Internet protocol requires replies (e.g., a response to an ICMP *ping* request), the outgoing packet is placed on a queue for the IP output process, *ipout*.

Computer Science & Engineering

Operating System Design

The Xinu Approach

Second Edition

With Intel and ARM Examples

Widely lauded for avoiding the typical black box approach found in other operating system textbooks, the first edition of this bestselling book taught readers how an operating system works and explained how to build it from the ground up.

Continuing to follow a logical pattern for system design, **Operating System Design: The Xinu Approach, Second Edition** removes the mystery from operating system design and consolidates the body of material into a systematic discipline. It presents a hierarchical design paradigm that organizes major operating system components in an orderly, understandable manner.

The book guides readers through the construction of a conventional process-based operating system using practical, straightforward primitives. It gives the implementation details of one set of primitives, usually the most popular set. Once readers understand how primitives can be implemented on conventional hardware, they can then easily implement alternative versions.

The text begins with a bare machine and proceeds step-by-step through the design and implementation of the Xinu operating system. The Xinu code runs on many hardware platforms. This second edition has been completely rewritten to contrast operating systems for RISC and CISC processors. Encouraging hands-on experimentation, the book provides updated code throughout and examples for two low-cost experimenter boards: BeagleBone Black from ARM and Galileo from Intel.

Features

- Covers topics in the order a designer follows when building a system
- Uses inexpensive embedded platforms from ARM and Intel
- Describes the main components in the design hierarchy
- Presents example software that illustrates the functions provided by each hierarchy level
- Gives readers the foundation to implement alternative versions of primitives
- Includes many practical examples and exercises that facilitate hands-on learning with the code
- Offers updated code and other information on the author's website

WITH VITALSOURCE® EBOOK

- Access online or download to your smartphone, tablet or PC/Mac
- Search the full text of this and other titles you own
- Make and share notes and highlights
- Copy and paste text and figures for use in your own documents
- Customize your view by changing font size and layout

CRC Press
Taylor & Francis Group
an informa business
www.crcpress.com

6000 Broken Sound Parkway, NW
Suite 300, Boca Raton, FL 33487
711 Third Avenue
New York, NY 10017
2 Park Square, Milton Park
Abingdon, Oxon OX14 4RN, UK

K25117
ISBN: 978-1-4987-1243-9
90000
9 781498 712439
www.crcpress.com