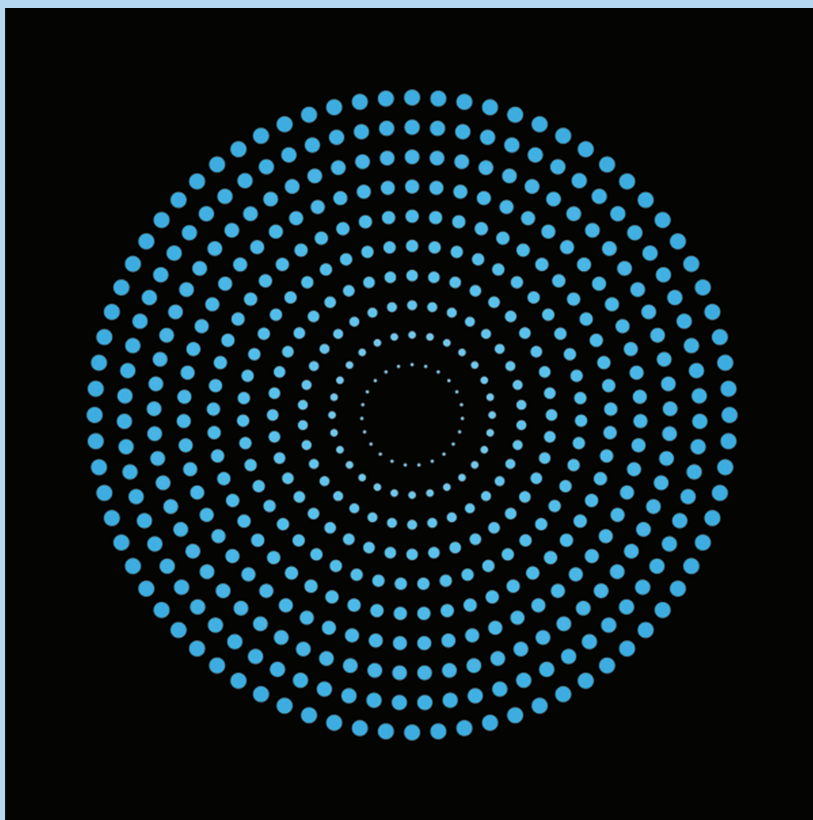


Operating System Design

The Xinu Approach

Linksys Version



Douglas Comer

Operating System Design

The Xinu Approach

Linksys Version

This page intentionally left blank

Operating System Design

The Xinu Approach

Linksys Version

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. 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 trademarks of MIPS Technologies, Inc. IBM is a registered trademark of International Business Machines. Linksys is a trademark of Cisco Systems. Mac is a trademark of Apple, Inc.

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

© 2012 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: 20110629

International Standard Book Number-13: 978-1-4398-8111-8 (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

This page intentionally left blank

Contents

Preface	xix
About The Author	xxiii
Chapter 1 Introduction and Overview	1
1.1 Operating Systems	1
1.2 Approach Used In The Text	3
1.3 A Hierarchical Design	3
1.4 The Xinu Operating System	5
1.5 What An Operating System Is Not	6
1.6 An Operating System Viewed From The Outside	7
1.7 Remainder Of The Text	8
1.8 Perspective	8
1.9 Summary	9
Chapter 2 Concurrent Execution And Operating System Services	11
2.1 Introduction	11
2.2 Programming Models For Multiple Activities	12
2.3 Operating System Services	13
2.4 Concurrent Processing Concepts And Terminology	13
2.5 Distinction Between Sequential And Concurrent Programs	15
2.6 Multiple Processes Sharing A Single Piece Of Code	17
2.7 Process Exit And Process Termination	19
2.8 Shared Memory, Race Conditions, And Synchronization	20
2.9 Semaphores And Mutual Exclusion	24
2.10 Type Names Used In Xinu	26
2.11 Operating System Debugging With Kputc And Kprintf	27
2.12 Perspective	28
2.13 Summary	28

Chapter 3 An Overview Of The Hardware and Run-Time Environment 31

- 3.1 *Introduction* 31
- 3.2 *Physical And Logical Organizations Of The E2100L* 32
- 3.3 *Processor Organization And Registers* 33
- 3.4 *Bus Operation: The Fetch-Store Paradigm* 34
- 3.5 *Direct Memory Access* 34
- 3.6 *The Bus Address Space* 35
- 3.7 *Contents Of Kernel Segments KSEG0 and KSEG1* 36
- 3.8 *Bus Startup Static Configuration* 37
- 3.9 *Calling Conventions And The Run-Time Stack* 38
- 3.10 *Interrupts And Interrupt Processing* 40
- 3.11 *Exception Processing* 41
- 3.12 *Timer Hardware* 42
- 3.13 *Serial Communication* 42
- 3.14 *Polled vs. Interrupt-Driven I/O* 43
- 3.15 *Memory Cache And KSEG1* 43
- 3.16 *Storage Layout* 44
- 3.17 *Memory Protection* 45
- 3.18 *Perspective* 45

Chapter 4 List and Queue Manipulation 49

- 4.1 *Introduction* 49
- 4.2 *A Unified Structure For Linked Lists Of Processes* 50
- 4.3 *A Compact List Data Structure* 51
- 4.4 *Implementation Of The Queue Data Structure* 53
- 4.5 *Inline Queue Manipulation Functions* 55
- 4.6 *Basic Functions To Extract A Process From A List* 55
- 4.7 *FIFO Queue Manipulation* 57
- 4.8 *Manipulation Of Priority Queues* 60
- 4.9 *List Initialization* 62
- 4.10 *Perspective* 63
- 4.11 *Summary* 64

Chapter 5 Scheduling and Context Switching 67

- 5.1 *Introduction* 67
- 5.2 *The Process Table* 68
- 5.3 *Process States* 71
- 5.4 *Ready And Current States* 72
- 5.5 *A Scheduling Policy* 72
- 5.6 *Implementation Of Scheduling* 73

5.7	<i>Implementation Of Context Switching</i>	76
5.8	<i>State Saved In Memory</i>	76
5.9	<i>Context Switch On A MIPS Processor</i>	77
5.10	<i>An Address At Which To Restart A Process</i>	80
5.11	<i>Concurrent Execution And A Null Process</i>	81
5.12	<i>Making A Process Ready And The Scheduling Invariant</i>	82
5.13	<i>Deferred Rescheduling</i>	83
5.14	<i>Other Process Scheduling Algorithms</i>	85
5.15	<i>Perspective</i>	86
5.16	<i>Summary</i>	86

Chapter 6 More Process Management

89

6.1	<i>Introduction</i>	89
6.2	<i>Process Suspension And Resumption</i>	89
6.3	<i>Self-Suspension And Information Hiding</i>	90
6.4	<i>The Concept Of A System Call</i>	91
6.5	<i>Interrupt Control With Disable And Restore</i>	93
6.6	<i>A System Call Template</i>	94
6.7	<i>System Call Return Values SYSERR And OK</i>	95
6.8	<i>Implementation Of Suspend</i>	95
6.9	<i>Suspending The Current Process</i>	97
6.10	<i>Suspend Return Value</i>	97
6.11	<i>Process Termination And Process Exit</i>	98
6.12	<i>Process Creation</i>	101
6.13	<i>Other Process Manager Functions</i>	106
6.14	<i>Summary</i>	108

Chapter 7 Coordination Of Concurrent Processes

111

7.1	<i>Introduction</i>	111
7.2	<i>The Need For Synchronization</i>	111
7.3	<i>A Conceptual View Of Counting Semaphores</i>	113
7.4	<i>Avoidance Of Busy Waiting</i>	113
7.5	<i>Semaphore Policy And Process Selection</i>	114
7.6	<i>The Waiting State</i>	115
7.7	<i>Semaphore Data Structures</i>	116
7.8	<i>The Wait System Call</i>	117
7.9	<i>The Signal System Call</i>	118
7.10	<i>Static And Dynamic Semaphore Allocation</i>	119
7.11	<i>Example Implementation Of Dynamic Semaphores</i>	120
7.12	<i>Semaphore Deletion</i>	121
7.13	<i>Semaphore Reset</i>	123

- 7.14 *Coordination Across Parallel Processors (Multicore)* 124
- 7.15 *Perspective* 125
- 7.16 *Summary* 125

Chapter 8 Message Passing 129

- 8.1 *Introduction* 129
- 8.2 *Two Types Of Message Passing Services* 129
- 8.3 *Limits On Resources Used By Messages* 130
- 8.4 *Message Passing Functions And State Transitions* 131
- 8.5 *Implementation Of Send* 132
- 8.6 *Implementation Of Receive* 134
- 8.7 *Implementation Of Non-Blocking Message Reception* 135
- 8.8 *Perspective* 135
- 8.9 *Summary* 136

Chapter 9 Basic Memory Management 139

- 9.1 *Introduction* 139
- 9.2 *Types Of Memory* 139
- 9.3 *Definition Of A Heavyweight Process* 140
- 9.4 *Memory Management In A Small Embedded System* 141
- 9.5 *Program Segments And Regions Of Memory* 142
- 9.6 *Dynamic Memory Allocation In An Embedded System* 143
- 9.7 *Design Of The Low-Level Memory Manager* 144
- 9.8 *Allocation Strategy And Memory Persistence* 144
- 9.9 *Keeping Track Of Free Memory* 145
- 9.10 *Implementation Of Low-Level Memory Management* 146
- 9.11 *Allocating Heap Storage* 148
- 9.12 *Allocating Stack Storage* 151
- 9.13 *Releasing Heap And Stack Storage* 153
- 9.14 *Perspective* 156
- 9.15 *Summary* 156

Chapter 10 High-Level Memory Management and Virtual Memory 159

- 10.1 *Introduction* 159
- 10.2 *Partitioned Space Allocation* 160
- 10.3 *Buffer Pools* 160
- 10.4 *Allocating A Buffer* 162
- 10.5 *Returning Buffers To The Buffer Pool* 164
- 10.6 *Creating A Buffer Pool* 165
- 10.7 *Initializing The Buffer Pool Table* 167

10.8	<i>Virtual Memory And Memory Multiplexing</i>	168
10.9	<i>Real And Virtual Address Spaces</i>	169
10.10	<i>Hardware For Demand Paging</i>	171
10.11	<i>Address Translation With A Page Table</i>	171
10.12	<i>Metadata In A Page Table Entry</i>	173
10.13	<i>Demand Paging And Design Questions</i>	173
10.14	<i>Page Replacement And Global Clock</i>	174
10.15	<i>Perspective</i>	175
10.16	<i>Summary</i>	175

Chapter 11 High-Level Message Passing **179**

11.1	<i>Introduction</i>	179
11.2	<i>Inter-Process Communication Ports</i>	179
11.3	<i>The Implementation Of Ports</i>	180
11.4	<i>Port Table Initialization</i>	181
11.5	<i>Port Creation</i>	183
11.6	<i>Sending A Message To A Port</i>	184
11.7	<i>Receiving A Message From A Port</i>	186
11.8	<i>Port Deletion And Reset</i>	188
11.9	<i>Perspective</i>	191
11.10	<i>Summary</i>	191

Chapter 12 Interrupt Processing **195**

12.1	<i>Introduction</i>	195
12.2	<i>The Advantage Of Interrupts</i>	196
12.3	<i>Interrupt Dispatching</i>	196
12.4	<i>Vectored Interrupts</i>	197
12.5	<i>Assignment Of Interrupt Vector Numbers</i>	197
12.6	<i>Interrupt Hardware</i>	198
12.7	<i>IRQ Limits And Interrupt Multiplexing</i>	199
12.8	<i>Interrupt Software And Dispatching</i>	199
12.9	<i>The Lowest Level Of The Interrupt Dispatcher</i>	202
12.10	<i>The High-Level Interrupt Dispatcher</i>	204
12.11	<i>Disabling Interrupts</i>	208
12.12	<i>Constraints On Functions That Interrupt Code Invokes</i>	208
12.13	<i>The Need To Reschedule During An Interrupt</i>	209
12.14	<i>Rescheduling During An Interrupt</i>	210
12.15	<i>Perspective</i>	211
12.16	<i>Summary</i>	211

Chapter 13 Real-Time Clock Management 215

- 13.1 Introduction 215*
- 13.2 Timed Events 216*
- 13.3 Real-Time Clocks And Timer Hardware 216*
- 13.4 Handling Real-Time Clock Interrupts 217*
- 13.5 Delay And Preemption 218*
- 13.6 Emulating A Real-Time Clock With A Timer 219*
- 13.7 Implementation Of Preemption 219*
- 13.8 Efficient Management Of Delay With A Delta List 220*
- 13.9 Delta List Implementation 221*
- 13.10 Putting A Process To Sleep 223*
- 13.11 Timed Message Reception 226*
- 13.12 Awakening Sleeping Processes 230*
- 13.13 Clock Interrupt Processing 231*
- 13.14 Clock Initialization 232*
- 13.15 Interval Timer Management 233*
- 13.16 Perspective 235*
- 13.17 Summary 235*

Chapter 14 Device-Independent Input and Output 239

- 14.1 Introduction 239*
- 14.2 Conceptual Organization Of I/O And Device Drivers 240*
- 14.3 Interface And Driver Abstractions 241*
- 14.4 An Example I/O Interface 242*
- 14.5 The Open-Read-Write-Close Paradigm 243*
- 14.6 Bindings For I/O Operations And Device Names 244*
- 14.7 Device Names In Xinu 245*
- 14.8 The Concept Of A Device Switch Table 245*
- 14.9 Multiple Copies Of A Device And Shared Drivers 246*
- 14.10 The Implementation Of High-Level I/O Operations 249*
- 14.11 Other High-Level I/O Functions 251*
- 14.12 Open, Close, And Reference Counting 255*
- 14.13 Null And Error Entries In Devtab 257*
- 14.14 Initialization Of The I/O System 258*
- 14.15 Perspective 262*
- 14.16 Summary 263*

Chapter 15 An Example Device Driver 267

- 15.1 Introduction 267*
- 15.2 The Tty Abstraction 267*

15.3	Organization Of A Tty Device Driver	269
15.4	Request Queues And Buffers	270
15.5	Synchronization Of Upper Half And Lower Half	271
15.6	Hardware Buffers And Driver Design	272
15.7	Tty Control Block And Data Declarations	273
15.8	Minor Device Numbers	276
15.9	Upper-Half Tty Character Input (ttyGetc)	277
15.10	Generalized Upper-Half Tty Input (ttyRead)	278
15.11	Upper-Half Tty Character Output (ttyPutc)	280
15.12	Starting Output (ttyKickOut)	281
15.13	Upper-Half Tty Multiple Character Output (ttyWrite)	282
15.14	Lower-Half Tty Driver Function (ttyInterrupt)	283
15.15	Output Interrupt Processing (ttyInter_out)	286
15.16	Tty Input Processing (ttyInter_in)	288
15.17	Tty Control Block Initialization (ttyInit)	295
15.18	Device Driver Control	297
15.19	Perspective	299
15.20	Summary	300

Chapter 16 DMA Devices And Drivers (Ethernet)

303

16.1	Introduction	303
16.2	Direct Memory Access And Buffers	303
16.3	Multiple Buffers And Rings	304
16.4	An Example Ethernet Driver Using DMA	305
16.5	Device Hardware Definitions And Constants	305
16.6	Rings And Buffers In Memory	309
16.7	Definitions Of An Ethernet Control Block	310
16.8	Device And Driver Initialization	313
16.9	Allocating An Input Buffer	318
16.10	Reading From An Ethernet Device	320
16.11	Writing To An Ethernet Device	322
16.12	Handling Interrupts From An Ethernet Device	324
16.13	Ethernet Control Functions	328
16.14	Perspective	330
16.15	Summary	330

Chapter 17 A Minimal Internet Protocol Stack

333

17.1	Introduction	333
17.2	Required Functionality	334
17.3	Simultaneous Conversations, Timeouts, And Processes	335
17.4	ARP Functions	336

17.5	<i>Definition Of A Network Packet</i>	346
17.6	<i>The Network Input Process</i>	347
17.7	<i>Definition Of The UDP Table</i>	351
17.8	<i>UDP Functions</i>	352
17.9	<i>Internet Control Message Protocol</i>	362
17.10	<i>Dynamic Host Configuration Protocol</i>	363
17.11	<i>Perspective</i>	368
17.12	<i>Summary</i>	368

Chapter 18 A Remote Disk Driver

371

18.1	<i>Introduction</i>	371
18.2	<i>The Disk Abstraction</i>	371
18.3	<i>Operations A Disk Driver Supports</i>	372
18.4	<i>Block Transfer And High-Level I/O Functions</i>	372
18.5	<i>A Remote Disk Paradigm</i>	373
18.6	<i>Semantics Of Disk Operations</i>	374
18.7	<i>Definition Of Driver Data Structures</i>	375
18.8	<i>Driver Initialization (rdsInit)</i>	381
18.9	<i>The Upper-Half Open Function (rdsOpen)</i>	384
18.10	<i>The Remote Communication Function (rdscmm)</i>	386
18.11	<i>The Upper-Half Write Function (rdsWrite)</i>	389
18.12	<i>The Upper-Half Read Function (rdsRead)</i>	391
18.13	<i>Flushing Pending Requests</i>	395
18.14	<i>The Upper-Half Control Function (rdsControl)</i>	396
18.15	<i>Allocating A Disk Buffer (rdsbufalloc)</i>	399
18.16	<i>The Upper-Half Close Function (rdsClose)</i>	400
18.17	<i>The Lower-Half Communication Process (rdsprocess)</i>	402
18.18	<i>Perspective</i>	407
18.19	<i>Summary</i>	407

Chapter 19 File Systems

411

19.1	<i>What Is A File System?</i>	411
19.2	<i>An Example Set Of File Operations</i>	412
19.3	<i>Design Of A Local File System</i>	413
19.4	<i>Data Structures For The Xinu File System</i>	413
19.5	<i>Implementation Of The Index Manager</i>	415
19.6	<i>Clearing An Index Block (lfibclear)</i>	420
19.7	<i>Retrieving An Index Block (lfibget)</i>	420
19.8	<i>Storing An Index Block (lfibput)</i>	421
19.9	<i>Allocating An Index Block From The Free List (lfiballoc)</i>	423
19.10	<i>Allocating A Data Block From The Free List (lfdalloc)</i>	424

19.11	Using The Device-Independent I/O Functions For Files	426
19.12	File System Device Configuration And Function Names	426
19.13	The Local File System Open Function (<i>lfsOpen</i>)	427
19.14	Closing A File Pseudo-Device (<i>lflClose</i>)	435
19.15	Flushing Data To Disk (<i>lfflush</i>)	436
19.16	Bulk Transfer Functions For A File (<i>lflWrite</i> , <i>lflRead</i>)	437
19.17	Seeking To A New Position In the File (<i>lflSeek</i>)	439
19.18	Extracting One Byte From A File (<i>lflGetc</i>)	440
19.19	Changing One Byte In A File (<i>lflPutc</i>)	442
19.20	Loading An Index Block And A Data Block (<i>lfsetup</i>)	443
19.21	Master File System Device Initialization (<i>lfsInit</i>)	447
19.22	Pseudo-Device Initialization (<i>lflInit</i>)	448
19.23	File Truncation (<i>lfltruncate</i>)	449
19.24	Initial File System Creation (<i>lfscreate</i>)	452
19.25	Perspective	454
19.26	Summary	455

Chapter 20 A Remote File Mechanism

459

20.1	Introduction	459
20.2	Remote File Access	459
20.3	Remote File Semantics	460
20.4	Remote File Design And Messages	460
20.5	Remote File Server Communication	468
20.6	Sending A Basic Message	470
20.7	Network Byte Order	472
20.8	A Remote File System Using A Device Paradigm	472
20.9	Opening A Remote File	474
20.10	Checking The File Mode	477
20.11	Closing A Remote File	478
20.12	Reading From A Remote File	479
20.13	Writing To A Remote File	482
20.14	Seek On A Remote File	485
20.15	Character I/O On A Remote File	486
20.16	Remote File System Control Functions	487
20.17	Initializing The Remote File Data Structure	491
20.18	Perspective	493
20.19	Summary	493

Chapter 21 A Syntactic Namespace

497

21.1	Introduction	497
21.2	Transparency And A Namespace Abstraction	497

21.3	<i>Myriad Naming Schemes</i>	498
21.4	<i>Naming System Design Alternatives</i>	500
21.5	<i>A Syntactic Namespace</i>	500
21.6	<i>Patterns And Replacements</i>	501
21.7	<i>Prefix Patterns</i>	501
21.8	<i>Implementation Of A Namespace</i>	502
21.9	<i>Namespace Data Structures And Constants</i>	502
21.10	<i>Adding Mappings To The Namespace Prefix Table</i>	503
21.11	<i>Mapping Names With The Prefix Table</i>	505
21.12	<i>Opening A Named File</i>	509
21.13	<i>Namespace Initialization</i>	510
21.14	<i>Ordering Entries In The Prefix Table</i>	513
21.15	<i>Choosing A Logical Namespace</i>	514
21.16	<i>A Default Hierarchy And The Null Prefix</i>	515
21.17	<i>Additional Object Manipulation Functions</i>	515
21.18	<i>Advantages And Limits Of The Namespace Approach</i>	516
21.19	<i>Generalized Patterns</i>	517
21.20	<i>Perspective</i>	518
21.21	<i>Summary</i>	518

Chapter 22 System Initialization

521

22.1	<i>Introduction</i>	521
22.2	<i>Bootstrap: Starting From Scratch</i>	521
22.3	<i>Operating System Initialization</i>	522
22.4	<i>Booting An Alternative Image On The E2100L</i>	523
22.5	<i>Xinu Initialization</i>	524
22.6	<i>System Startup</i>	527
22.7	<i>Transforming A Program Into A Process</i>	531
22.8	<i>Perspective</i>	532
22.9	<i>Summary</i>	532

Chapter 23 Exception Handling

535

23.1	<i>Introduction</i>	535
23.2	<i>Exceptions, Traps, And Illegal Interrupts</i>	535
23.3	<i>Implementation Of Panic</i>	536
23.4	<i>Perspective</i>	537
23.5	<i>Summary</i>	538

Chapter 24 System Configuration 541

- 24.1 Introduction 541
- 24.2 The Need For Multiple Configurations 541
- 24.3 Configuration In Xinu 542
- 24.4 Contents Of The Xinu Configuration File 543
- 24.5 Computation Of Minor Device Numbers 545
- 24.6 Steps In Configuring A Xinu System 546
- 24.7 Perspective 547
- 24.8 Summary 547

Chapter 25 An Example User Interface: The Xinu Shell 549

- 25.1 Introduction 549
- 25.2 What Is A User Interface? 550
- 25.3 Commands And Design Principles 550
- 25.4 Design Decisions For A Simplified Shell 551
- 25.5 Shell Organization And Operation 551
- 25.6 The Definition Of Lexical Tokens 552
- 25.7 The Definition Of Command-Line Syntax 552
- 25.8 Implementation Of The Xinu Shell 553
- 25.9 Storage Of Tokens 555
- 25.10 Code For The Lexical Analyzer 556
- 25.11 The Heart Of The Command Interpreter 561
- 25.12 Command Name Lookup And Builtin Processing 569
- 25.13 Arguments Passed To Commands 570
- 25.14 Passing Arguments To A Non-Builtin Command 571
- 25.15 I/O Redirection 575
- 25.16 An Example Command Function (sleep) 575
- 25.17 Perspective 577
- 25.18 Summary 578

Appendix 1 Porting An Operating System 580

- A1.1 Introduction 580
- A1.2 Motivation: Evolving Hardware 581
- A1.3 Steps Taken When Porting An Operating System 581
- A1.4 Programming To Accommodate Change 587
- A1.5 Summary 589

Appendix 2 Xinu Design Notes	590
<i>A2.1 Introduction</i>	590
<i>A2.2 Overview</i>	590
<i>A2.3 Xinu Design Notes</i>	591
<i>A2.4 Xinu Implementation</i>	592
<i>A2.5 Major Concepts And Implementation</i>	593
 Index	 596

Chapter Contents

- 3.1 Introduction, 31
- 3.2 Physical And Logical Organizations Of The E2100L, 32
- 3.3 Processor Organization And Registers, 33
- 3.4 Bus Operation: The Fetch-Store Paradigm, 34
- 3.5 Direct Memory Access, 34
- 3.6 The Bus Address Space, 35
- 3.7 Contents Of Kernel Segments KSEG0 and KSEG1, 36
- 3.8 Bus Startup Static Configuration, 37
- 3.9 Calling Conventions And The Run-Time Stack, 38
- 3.10 Interrupts And Interrupt Processing, 40
- 3.11 Exception Processing, 41
- 3.12 Timer Hardware, 42
- 3.13 Serial Communication, 42
- 3.14 Polled vs. Interrupt-Driven I/O, 43
- 3.15 Memory Cache And KSEG1, 43
- 3.16 Storage Layout, 44
- 3.17 Memory Protection, 45
- 3.18 Perspective, 45

3

An Overview Of The Hardware and Run-Time 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. The example system in this text runs on a small embedded hardware platform, the E2100L Linksys wireless router. We chose the Linksys router because it offers simplicity, a well-known instruction set, availability, and low cost. The system is small enough to allow readers to understand most of the hardware, and sufficiently complex to illustrate how an operating system works on a general-purpose system. Finally, the E2100L allows programmers to download and run code without requiring a sophisticated hardware lab and without replacing ROM chips.

The remainder of the chapter introduces the Linksys hardware, describing pertinent features of the processor, memory, and I/O devices. The chapter explains the architecture, memory address space, the run-time stack, the interrupt mechanism, and device addressing. Although the details refer to the E2100L, the basic concepts apply broadly to most computer systems.

3.2 Physical And Logical Organizations Of The E2100L

Physically, a Linksys router consists of a small self-contained box that uses a separate power cube. Because Linksys markets the E2100L as a consumer product, the router comes completely assembled. Furthermore, most of the major components are contained in a single VLSI chip that is known as a *System On a Chip (SoC)*.

Although the circuit board contains pins for a serial interface, no external connection is supplied. Before the serial interface can be used, an inexpensive serial converter must be connected to pins on the board. 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>

Logically, the E2100L follows the same overall architecture as most general-purpose computer systems. The components on the SoC include a processor, co-processor, memory interface, and I/O device interfaces. The I/O devices of interest include wired and wireless network devices.

One wired network interface connects to a controller that acts as a hub with four RJ-45 sockets that can connect to local computers.[†] The other wired network interface, intended for an Internet connection, attaches to a single RJ-45 socket. An internal bus, known as a *system backplane*, provides a central interconnect that allows components to interact. Figure 3.1 illustrates the conceptual organization.

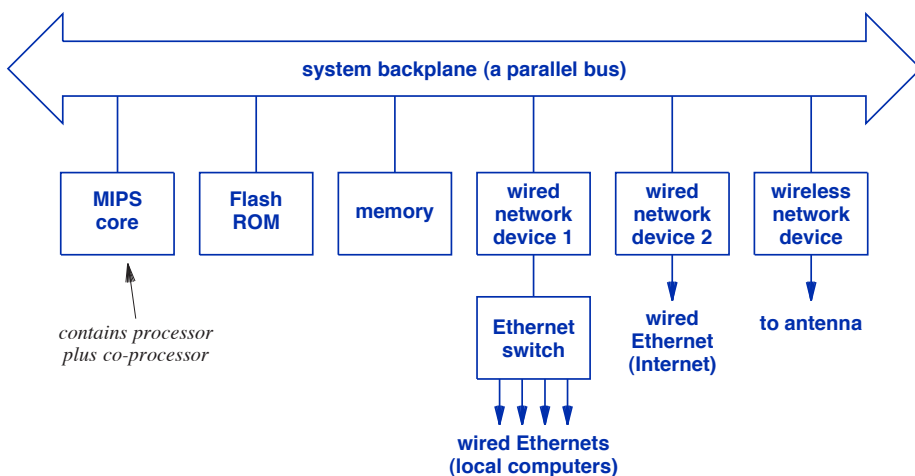


Figure 3.1 The logical organization of major components in the E2100L.

[†]Hardware components on the chip can be reconfigured to provide alternative logical organizations. The chapter and the remainder of the text describe the default hardware configuration that is available when the device is powered on without giving details about alternatives.

The next sections consider features of the E2100L that are 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.

3.3 Processor Organization And Registers

Like many embedded systems, the E2100L uses a *RISC*[†] processor. The processor implements a *MIPS* instruction set. Except for a few special cases, an operating system does not need to focus on the instruction set because a compiler generates the necessary code.

The processor contains thirty-two *general-purpose registers*. Each register is 32 bits wide, and can hold an integer, an address, or four 8-bit characters. As in most RISC processors, many MIPS operations take arguments in registers and leave the result of the operation in a register. Despite being classified as general-purpose, individual registers are assigned specific uses by the compiler. For example, one of the registers is used as a stack pointer that is changed to allocate space for an activation record whenever a function call occurs. Figure 3.2 lists the registers by giving the name used to reference a register and the meaning assigned.

Name	Meaning
0	Always contains zero
AT	Assembler temporary (reserved for assembler to use)
V0 and V1	Return values from a function call
A0 – A3	Argument registers for first 4 arguments
T0 – T9	Temporary (i.e., not preserved) across a function call
S0 – S9	Saved (i.e., preserved) across a function call
K0 and K1	Kernel registers used by interrupt hardware
SP	Stack pointer (stack grows downward)
RA	Return address during a function call

Figure 3.2 The general-purpose registers and the meaning of each.

[†]RISC stands for Reduced Instruction Set Computer.

A co-processor contains a set of special-purpose registers that support many extra features. For example, because a RISC processor cannot perform division in one cycle, the co-processor handles 64-bit division. Consequently, the co-processor has a pair of registers that each store half of a 64-bit value. One register stores the high 32 bits and the other stores the low 32 bits. Similarly, the co-processor has registers that store: the current *interrupt mask* (telling whether interrupts are enabled), a return address for interrupts or exceptions, and debugging information. Chapter 12 on interrupt processing will show how an operating system uses specific co-processor registers.

3.4 Bus Operation: The Fetch-Store Paradigm

The bus, known as a *system backplane*, provides the primary path between the processor and other components, namely the memory, I/O devices, and other interface controllers. The system backplane uses a *fetch-store paradigm* typical of most computer system buses. For example, when it needs to access memory, the processor places an 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 system uses *memory-mapped I/O*, which means that each I/O device is assigned a set of addresses in the bus address space, and the processor uses the same fetch-store paradigm to communicate with I/O devices as with memory. As we will see, 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 loads a value from the address into a register.

3.5 Direct Memory Access

Some I/O devices on the E2100L 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 CPU frequently nor does it require the processor to control 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 code while the device operates.

As an example, consider how a network device uses DMA. To receive a packet, the operating system allocates a buffer in memory and starts the network device. When a packet arrives, the device hardware transfers the packet directly into the buffer in memory and interrupts the processor. To send a packet, the operating system places the packet in a buffer in memory and starts the device. The device fetches the packet from the buffer in memory and transmits the packet on the network.

In addition to single packet transfers, DMA hardware on the E2100L allows the 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 them. Later chapters explain more DMA details, and the example code illustrates how a device driver in the operating system allocates I/O buffers and controls DMA operations.

3.6 The Bus Address Space

The system backplane uses a 32-bit bus address space, with addresses ranging from 0x00000000 through 0xFFFFFFFF. Some of the addresses in the address space correspond to memory, some to FlashROM, and others to I/O devices. To accommodate operating systems that support memory protection and virtual memory, the address space is divided in half, with the lower half (addresses 0x00000000 through 0x7FFFFFFF) comprising a *user space* and the upper half (addresses 0x80000000 through 0xFFFFFFFF) comprising the *kernel space*, which is further divided into *segments*. Figure 3.3 illustrates the organization.

Memory on the E2100L 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, the E2100L does not contain 4 Gbytes of memory. Instead, the physical memory occupies 16 Mbytes that appear to be repeated in the address space. The next section explains the replication.

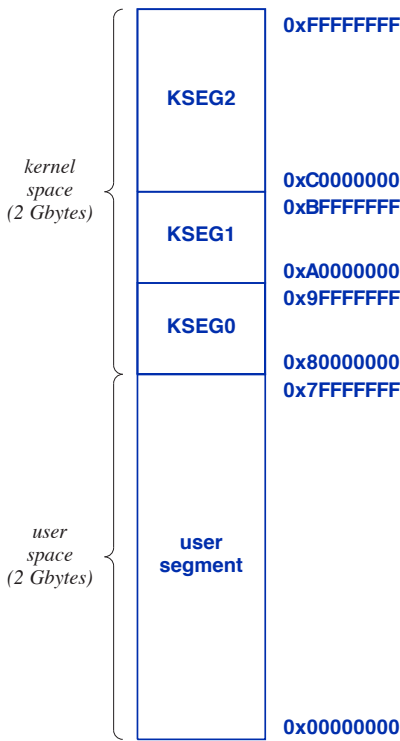


Figure 3.3 The division of the bus address space into user and kernel areas, with further division into segments.

3.7 Contents Of Kernel Segments KSEG0 and KSEG1

Segments KSEG0 and KSEG1 are fundamental to the operating system, and have special meaning. The operating system resides in KSEG0. The highest addresses of the segment are reserved for I/O devices; lower addresses correspond to physical memory. Figure 3.4 illustrates the layout of KSEG0.

Although KSEG0 contains addresses for 512 Mbytes, the physical memory on the E2100L comprises only 16 Mbytes. The physical memory appears to be replicated through the address space. For example, physical memory occupies relative addresses 0x00000000 through 0x000FFFFF. The same physical memory appears again in locations 0x00100000 through 0x001FFFFF. That is, the hardware ignores high-order bits of an address when mapping the address to memory.[†]

[†]Although embedded systems tend to ignore high-order address bits, many computer systems restrict addresses to physical memory, and classify other references as errors.

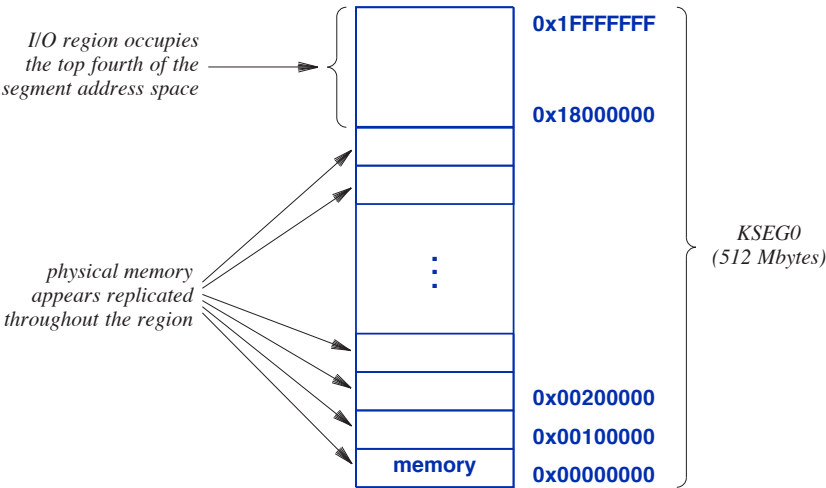


Figure 3.4 The KSEG0 address space.

The important point is:

Referencing an address beyond physical memory may not cause an error to occur. Instead, high-order bits of the address are ignored, and the address may map into a valid reference as if the physical memory is replicated.

3.8 Bus Startup Static Configuration

Desktop computers contain complex bus hardware that allows a processor to discover components that attach to the bus. Each component is assigned a unique identifier that specifies the vendor and the specific type. At startup, an operating system probes the bus to determine which components are present, and each component returns its ID. The mechanism makes it possible for an operating system to configure itself dynamically, allowing a copy of the system to run on a variety of hardware platforms.

Unlike desktop systems, embedded systems often use *static configuration*. That is, the complete hardware configuration is known when an operating system is designed, and the system does not discover hardware at startup, nor does it reconfigure. Chapter 24 discusses system configuration in more detail, and shows an example of static configuration.

3.9 Calling Conventions And The Run-Time Stack

Function invocation forms a key aspect of operating systems. Application programs use function calls to invoke operating system services such as creating concurrent processes and performing I/O. As it switches context from one process to another, the operating system must manage separate sets of function calls. The following paragraphs define key concepts 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 many choices to a compiler writer. We will see that because it calls functions to process interrupts and switches from one running process to another, the operating system must understand and follow the same conventions as the compiler.

Run-time stack. A statically scoped language, such as C, uses a *run-time 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. Also by convention, a stack grows downward from higher memory addresses to lower memory addresses. Thus, when a function call occurs, the stack grows downward in memory to hold the activation record for the function call.

Arguments. When a function is called, the caller supplies a set of actual *arguments* that correspond to *parameters*. Most RISC architectures arrange a fixed number of arguments to be passed in registers and the remaining arguments to be passed in memory. The example code uses registers A0 through A3 to pass the first four arguments; arguments beyond the first four are placed in the activation record on the stack.

Stack frame contents. A compiler computes the size needed for a stack frame, and handles assigning space to each local variable. However, an operating system needs to know exactly how arguments are stored. In the example code in the text, the highest word in each stack frame is used to store a return address, the lowest four words are reserved as an argument save area, and successive words beyond the argument save area contain arguments beyond the first four arguments. Figure 3.5 illustrates the format.

Interestingly, the argument save area that occupies the lowest four words of a stack frame for function, *f*, is not used by *f*. Instead, the area is reserved for use by functions that *f* calls. In the figure, if function *X* calls function *Y*, the lowest four words of the stack for function *X* will be used by function *Y*.

To understand why an argument save area is needed, observe that the first four arguments are passed in registers A0 through A3. For example, suppose function *X* calls function *Y*, and passes an argument *q*. Register A0 will contain the value of *q* when *Y* begins execution. If function *Y* calls function *Z* and passes argument *r*, register A0 will be used. That is, to pass argument *r* to function *Z*, *Y* must overwrite the value *q* that it received from *X*. To insure that *q* is not lost, *Y* saves the value of *q* on the stack before calling *Z*, and then restores the value after the call returns. In theory, *Y* could save A0

at any location in memory — saving arguments in the caller’s stack frame is merely a convention adopted by the compiler being used.

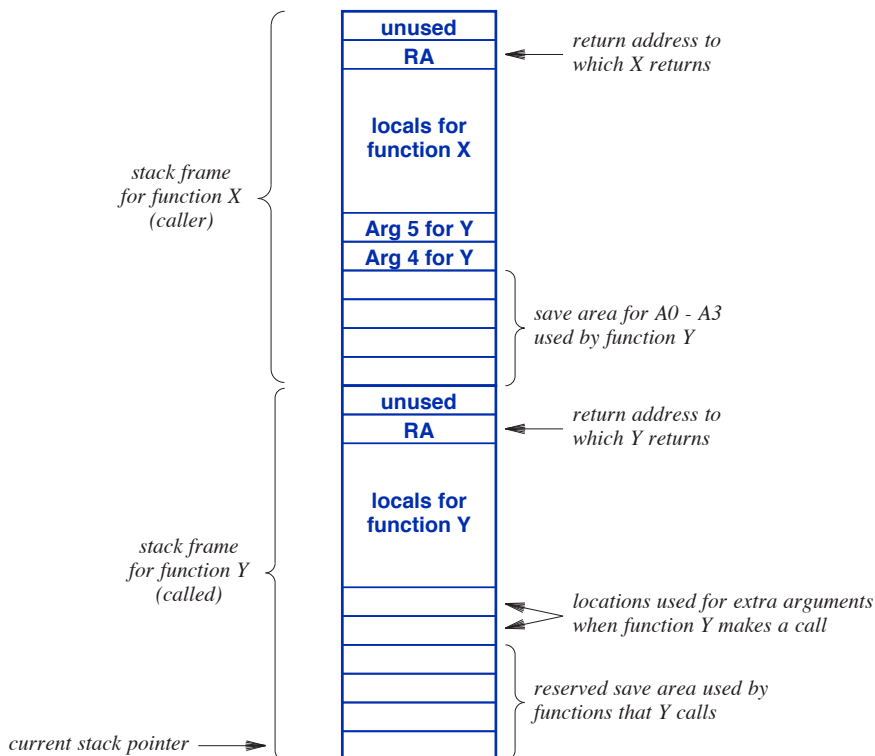


Figure 3.5 Format of two stack frames when function X calls function Y.

As a further convention, the *frame pointer* register is not used. Instead, the compiler computes the exact size of the stack frame that will be needed for each function (including the four word argument save area), and arranges code to decrement the stack pointer by the appropriate size when a function is called. Because a RISC processor does not include instructions that push values onto the stack, the compiler can compute the address of each local variable as a fixed offset beyond the stack pointer.

To summarize:

When referencing local variables or arguments on the run-time stack, the operating system follows the calling conventions used by the compiler. In the example code throughout the text, all references are given as offsets beyond the stack pointer register.

3.10 Interrupts And Interrupt Processing

Modern processors provide mechanisms that allow external I/O devices to *interrupt* computation. Often, a processor has a 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 CPU 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 contain the hardware to handle interrupts. Instead, a co-processor performs much of the basic bus interaction and interrupt processing on behalf of the main processor.

For example, when a device interrupts, a MIPS processor performs three key steps:

- Sets a control bit that disables further interrupts
- Records the address of the instruction that is about to execute
- Jumps to the reserved location 0x80000180

The first step insures that while it is processing an interrupt from one device, the operating system will not be interrupted by another device. The second step provides a way for an operating system to return to normal execution once the interrupt has been processed. The third step allows the operating system to gain control whenever an interrupt occurs. Before an interrupt occurs, the operating system must store interrupt processing code at the reserved location (0x80000180). The compiler and loader start the operating system at location 0x80010000 (i.e., above the interrupt location), which guarantees that code can be stored in the reserved location without affecting other values in the operating system. At system startup, our example system stores a few instructions starting at the reserved location, which causes the processor to jump to an operating system interrupt function whenever an interrupt occurs.

While it manipulates global data structures and I/O queues, an operating system must prevent interrupts from occurring. The hardware supplies mechanisms to control interrupts. For example, one of the registers in the co-processor is an *interrupt mask* that the operating system uses to specify which devices are permitted to interrupt. Each bit in the mask corresponds to an interrupt source in the system; a source can consist of devices on the system bus, internal timers, or can arise from the execution of special processor opcodes. The initial value of all bits is zero, which means the corresponding source cannot interrupt. When it starts an I/O device, the operating system sets the corresponding mask bit to one, which allows the device to interrupt.

In addition to an individual bit for each device, the interrupt mask contains a *global interrupt status* bit. Setting the global bit to zero prevents all interrupts, despite the

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

individual device bits. We will see that functions *disable* and *restore* manipulate the global status bit to allow the operating system to prevent interrupts temporarily and later re-enable interrupts.

Figure 3.6 lists the co-processor registers associated with interrupt processing and describes the purpose of each.

Register	Purpose
STATUS	Interrupt status, including an EXL bit that specifies whether an interrupt is currently being processed, a global bit that tells whether all interrupts are disabled, and one interrupt enable bit for each source
CAUSE	Bits that identify the cause of an exception or the source of an interrupt
EPC	Exception Program Counter that specifies the address at which normal processing should resume after an interrupt has been handled

Figure 3.6 Hardware registers in the co-processor associated with interrupt processing.

The processor interrogates co-processor control registers to determine which device caused the interrupt, and then interacts with the device. Once the interrupt has been handled, the processor can execute an *interrupt return* instruction to resume normal processing. Later chapters provide examples of interrupt processing and the steps taken to interact with various devices.

3.11 Exception Processing

Although exceptions are generated by the processor rather than by a separate I/O device, the MIPS hardware integrates exception handling with interrupt handling. That is, an exception causes the hardware to take the same steps as an interrupt: setting the EXL bit to disable further interrupts, recording the address of the instruction that caused the exception in the *EPC* register, and jumping to location 0x80000180.

One minor difference occurs between the way the processor handles interrupts and exceptions. When an interrupt occurs, the processor has completed executing one instruction, and is about to execute another. Thus, the hardware uses the *EPC* register to record the address of the next instruction to be executed. When an exception occurs, however, an instruction is currently being executed when the exception arises. Thus, the hardware uses the *EPC* register to record the address of the instruction being execut-

ed. When the processor returns from the exception, the instruction will be restarted. For example, if a page fault occurs, the exception handler reads the missing page from memory, and returns to the same point at which the exception occurred to restart the instruction that caused the fault.

3.12 Timer Hardware

In addition to external I/O devices, the E2100L hardware includes a *timer device*. When it expires, the timer generates an interrupt, which means that once it has enabled timer interrupts, the processor must be prepared to handle an interrupt.

On some embedded systems, all timer functions are implemented using a real-time hardware clock that generates interrupts regularly (e.g., 60 times per second). On the E2100L, however, the timer consists of two registers that the processor can load:

- **Counter:** the counter register is set to an initial value
- **Limit:** the limit register specifies the length of time to wait

The hardware uses the CPU clock, and increments the counter register once per cycle. When the counter reaches the limit, the hardware generates a timer interrupt.

Both the real-time clock approach and the E2100L's timer mechanism have advantages. The chief advantage of the timer mechanism arises from fewer interrupts — unlike a real-time clock, which interrupts continually, a timer only interrupts when a preset timeout has occurred. The real-time clock approach has the advantage of being able to relate interrupts directly to actual time instead of processor clock cycles. Of course, if one knows the processor clock rate, it is possible to convert to real time. Unfortunately, the calculation depends on the CPU speed, which means the operating system cannot be ported to a faster processor without changing the constant used in the conversion.

3.13 Serial Communication

Serial communication devices are among the simplest I/O devices available, and have been used on computers for decades. The E2100L contains an RS-232 serial communication device that is used as a system console. As with most serial devices, the serial hardware on the E2100L handles both input and output (i.e., both transmission and reception of characters). When an interrupt occurs, the processor must examine a device register to determine whether the output side has completed transmission or the input side has received a character. Chapter 15 examines serial interrupts in detail.

3.14 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. For example, consider displaying a startup message for a user before the operating system has initialized interrupts and I/O. Also consider a programmer who needs to debug new I/O code. 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.

3.15 Memory Cache And KSEG1

Recall that although KSEG1 follows KSEG0 in the address space, memory locations appear to be replicated. Thus, the first byte of KSEG1 points to the same physical memory as the first byte of KSEG0.

Despite the appearance of replication, the hardware enforces an important distinction between the two memory segments:

When the processor references an address in KSEG0, the reference goes to the L1 memory cache before being passed to the bus (i.e., system backplane); when the processor references an address in KSEG1, the reference is passed directly to the bus.

For normal data references, a memory cache provides an important optimization — if the processor makes multiple references to an address within a short time span, the cache hardware returns the value faster than a memory reference. When performing I/O, however, a cache produces incorrect results. For example, consider the code that checks the status of a device when using a polled I/O paradigm. If a cache remembers and returns the previous value without accessing the device, the processor will not receive an accurate report of the device status. Thus, an operating system adheres to the following straightforward rule:

To avoid obtaining stale values from the memory cache, each I/O reference, including addresses specified for DMA, must use KSEG1.

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 3.7 illustrates the conceptual organization.

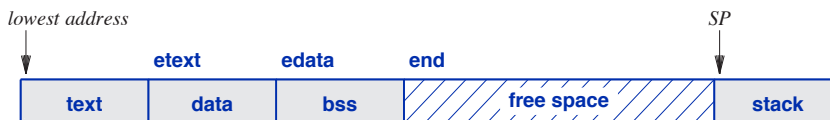


Figure 3.7 Illustration of memory segments created by a C compiler.

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. Although all processes share the text, data, and bss segments, a separate stack segment must be allocated for each process. If three processes are executing, the stacks are allocated contiguously from the highest memory address downward as Figure 3.8 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 3.8 Illustration of memory with stack segments for three processes.

3.17 Memory Protection

The memory hardware available on the E2100L has multiple segments that can be used to provide protection for an operating system. 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 insure that an application only receives carefully controlled services.

Like most embedded systems, our example system avoids the complexity and run-time overhead of memory protection. Instead the code runs completely in KSEG0, and each process runs with kernel privilege. 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 run-time stack will overwrite data in another process’s stack. A later chapter discusses one technique software can use to help detect overflow.

3.18 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.

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 independent timers, each with its own interrupt source. 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.

Chapter Contents

- 9.1 Introduction, 139
- 9.2 Types Of Memory, 139
- 9.3 Definition Of A Heavyweight Process, 140
- 9.4 Memory Management In A Small Embedded System, 141
- 9.5 Program Segments And Regions Of Memory, 142
- 9.6 Dynamic Memory Allocation In An Embedded System, 143
- 9.7 Design Of The Low-Level Memory Manager, 144
- 9.8 Allocation Strategy And Memory Persistence, 144
- 9.9 Keeping Track Of Free Memory, 145
- 9.10 Implementation Of Low-Level Memory Management, 146
- 9.11 Allocating Heap Storage, 148
- 9.12 Allocating Stack Storage, 151
- 9.13 Releasing Heap And Stack Storage, 153
- 9.14 Perspective, 156
- 9.15 Summary, 156

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.

In most embedded systems, main memory consists of a contiguous set of locations with addresses 0 through $N-1$; the code and data remain resident in memory. A small embedded system can use two memory technologies:

- *Read-Only Memory (ROM)*, which may include Flash, used to store program code and constants
- *Random Access Memory (RAM)* used to hold variables that can change during execution

In terms of the addresses used, the two types of memory occupy separate locations. For example, addresses 0 through $K-1$ might correspond to ROM, and addresses K through $N-1$ might correspond to RAM.[†]

Some systems further distinguish regions of memory by using specific types of memory technologies. For example, RAM can 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

Large operating systems provide protection mechanisms that prevent an application from reading or modifying areas of memory that have been assigned to another application. In particular, Chapter 10 discusses how a separate virtual address space can be assigned to each computation. 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 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 use a hybrid approach that includes a *lightweight process abstraction* (i.e., *threads of execution*). Instead of a single process executing in an address space, the operating system permits a user to create multiple threads that each execute in the address space.

In a hybrid system, a thread is similar to a process in Xinu. Each thread has a separate run-time stack used to hold local variables and function calls; the stacks are allocated from the data area in the heavyweight process's address space. All the threads

[†]Usually, K and N are each a power of 2.

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 a hybrid system.

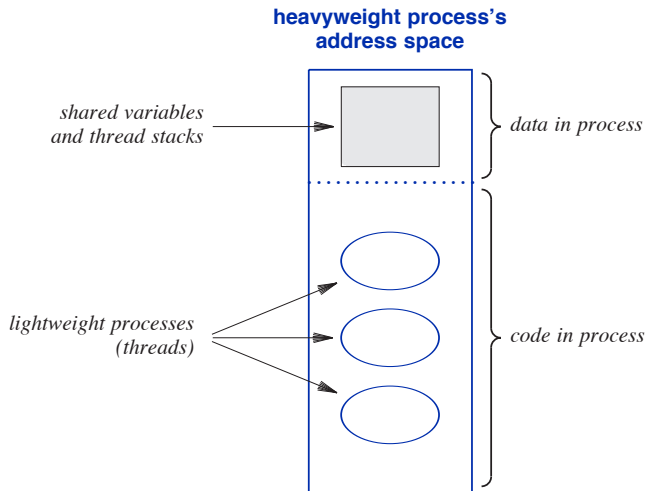


Figure 9.1 Illustration of a heavyweight process abstraction concept with multiple lightweight processes (threads) sharing the address space.

9.4 Memory Management In A Small Embedded System

The largest embedded systems, such as those used in video game consoles, have secondary storage and the memory management hardware needed to support virtual address spaces. 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

When Xinu is compiled for a small embedded system, the memory image is divided into four contiguous regions:

- Text segment
- Data segment
- Bss segment
- Free space

Text segment. The text segment, which begins at memory location zero, contains compiled code for each of the functions in the memory image. The text segment may also contain constants (e.g., string constants). 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 run-time.

Data segment. The data segment, which follows the text segment, contains storage for all global variables that are assigned an initial value. Values in the data segment may be accessed or modified (i.e., read or written).

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. Memory beyond the bss segment is considered *free* (i.e., unallocated) when execution begins. The next sections describe how the operating system uses the free space.

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.



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

[†]External symbol names have an underscore prepended by the loader. Thus, *etext* becomes *_etext*.

use one of the external symbols to reference a location and should not load a value. For example, the text segment occupies memory locations 0 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 size of the free space? The operating system must either be configured with the highest physical memory address or must probe addresses at startup.[†] At startup, our system places the highest valid memory address in global variable *maxheap*. Thus, to compute the initial size of the free space, the memory management code can take the difference between the address in variable *maxheap* and the value *&end*.

9.6 Dynamic Memory Allocation In An Embedded System

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 other two types of memory are:

- 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 to hold dynamically allocated variables that 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 address of free space. Second, whenever a process requests heap storage, Xinu allocates the necessary amount of space from the low end of free space. 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.

[†]Because physical memory is replicated in the E2100L address space, probing memory is difficult.

9.7 Design Of The Low-Level Memory Manager

A set of functions and data structures are used to manage free memory. The low-level memory manager provides four 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

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, and places the stack address in a *CONTEXT* area on top of the stack. Later, when the process becomes current, the context switch accesses the *CONTEXT* area and loads the address of the stack into the stack pointer register. 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 blocks from the lowest address of the free space.

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 one hand, demand can exceed the available space. On the other hand, the free space can become *fragmented* into small, discontinuous pieces that are each too small to satisfy a request. The next 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. Initially, the list contains only one item that corresponds to the block of memory between the end of the program and the highest point in memory. 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. 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 A conceptual list of free memory blocks.

A memory manager must examine each transaction carefully to avoid generating an arbitrarily long list of 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 where there is one entry that corresponds to all free memory between the end of the program and the highest memory address.

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 conceptual list in Figure 9.4 shows two fields associated with each entry: the 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.

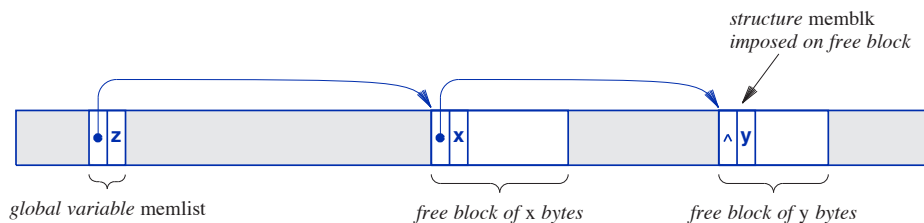


Figure 9.5 Illustration of a free memory list that contains two memory blocks.

Structure *memblk*, defined in file *memory.h*, gives the shape of a structure that can be imposed on each free node. In struct *memblk*, field *mnxt* 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 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. However, the *mlength* field (used to store the size of a block) is not meaningful in variable *memlist* 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 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: a memory manager cannot store a free block of less than 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 allow the memory management routines to insure 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. Two inline functions, *roundmb* and *truncmb*, perform the task. 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 once: the initial size of free space must be truncated rather than rounded. The key point is:

Rounding all requests to multiples of the memblk structure insures that each request satisfies the constraint and guarantees that no free block will ever be too small to link into the free list.

File *memory.h* contains declarations related to memory management, including definitions of the two inline functions, *roundmb* and *truncmb*. 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.

```

/* memory.h - roundmb, truncmb, freestk */

#define PAGE_SIZE      4096
#define MAXADDR        0x02000000    /* 160NL has 32MB RAM */

/*-----
 * 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 {
    struct memblk *mnext;    /* see roundmb & truncmb */
    uint32 mlength;          /* ptr to next free memory blk */
};                          /* size of blk (includes memblk) */

extern struct memblk memlist; /* head of free memory list */
extern void *maxheap;         /* max free memory address */
extern void *minheap;         /* address beyond loaded memory */

/* added by linker */

extern int end;               /* end of program */
extern int edata;             /* end of data segment */
extern int etext;             /* end of text segment */

```

9.11 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 *)SYSEERR;
    }

    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. *Getmem* 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 insure 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, *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 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* merely 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. 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 perform the division, *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* does not produce the desired result because C performs pointer arithmetic. To force C to use integer arithmetic instead of pointer arithmetic, *curr* is changed to an unsigned integer with a cast (i.e., *(uint32)curr*) before adding *nbytes*. Once a sum has been computed, the result is changed back into a pointer to a memory block using another cast. 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. 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.

9.12 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 *next* 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 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;

    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));
}

```

9.13 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;
    }

    /* Insure 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 148.

9.14 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 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, discontinuous 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.

9.15 Summary

Large operating systems offer complex memory management schemes that allow the demand for memory to exceed the physical memory size. Such systems store information on secondary storage, and move it to main memory when a reference occurs. Sophisticated memory management systems support multiple virtual address spaces that allow each application to address memory starting at location zero; each virtual address space maps to a different region of physical memory, meaning that the system protects a given application from other applications. 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 virtual space. Paging is the most widely used technology used to provide virtual address spaces and multiplex them onto physical memory.

Small embedded systems usually keep all code and data in physical memory. An 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.

The low-level memory manager in Xinu maintains a linked list of 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.

The 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 program-

mer must plan carefully to avoid starvation. A higher-level memory manager discussed in the next chapter illustrates how memory can be partitioned into regions and how a set of processes can block waiting for memory to become available.

EXERCISES

- 9.1 An early version of the low-level memory manager had no provision for returning blocks of memory to the free list. Speculate about embedded systems: are *freemem* and *freestk* necessary? Why or why not?
- 9.2 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*?
- 9.3 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.
- 9.4 Can the memory management system described here allocate arbitrarily small amounts of memory? Why or why not?
- 9.5 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.

Chapter Contents

- 18.1 Introduction, 371
- 18.2 The Disk Abstraction, 371
- 18.3 Operations A Disk Driver Supports, 372
- 18.4 Block Transfer And High-Level I/O Functions, 372
- 18.5 A Remote Disk Paradigm, 373
- 18.6 Semantics Of Disk Operations, 374
- 18.7 Definition Of Driver Data Structures, 375
- 18.8 Driver Initialization (rdsInit), 381
- 18.9 The Upper-Half Open Function (rdsOpen), 384
- 18.10 The Remote Communication Function (rdscomm), 386
- 18.11 The Upper-Half Write Function (rdsWrite), 389
- 18.12 The Upper-Half Read Function (rdsRead), 391
- 18.13 Flushing Pending Requests, 395
- 18.14 The Upper-Half Control Function (rdsControl), 396
- 18.15 Allocating A Disk Buffer (rdsbufalloc), 399
- 18.16 The Upper-Half Close Function (rdsClose), 400
- 18.17 The Lower-Half Communication Process (rdsprocess), 402
- 18.18 Perspective, 407
- 18.19 Summary, 407

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*: data persists even if power is removed.
- *Block-oriented*: the interface provides the ability to read or write fixed-size blocks of data.
- *Multi-use*: a block can be read and written many times.
- *Random-access*: blocks can be accessed 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 CPU. 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 disk 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 designated location in memory.
- *Store*: Copy the contents of memory to a specified block on the disk.
- *Seek*: Move to a specified block on the disk. The seek option is most important for electro-mechanical devices (i.e., a magnetic disk) because it can be used as an optimization that positions the disk head where it will be needed in the future. Thus, as solid state disks become widely used, seek may become less important.

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 provides block transfer, it makes sense to design an interface that allows *read* and *write* to transfer an entire block. The question becomes how to include a block specification in the existing high-level I/O operations. We might use *seek*: 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 an underlying block size of 4K bytes, the hardware presents an interface that uses 512-byte blocks.

Our driver will supply basic operations: *read*, which copies a single block from the disk into memory; *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., insure that all cached data is written to the disk).

18.5 A Remote Disk Paradigm

Because the E2100L hardware does not include a local disk, we will examine software that follows 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.

Driver software for the remote disk will be organized in the same general way as a driver for a traditional disk: driver functions are partitioned into an upper half and a lower half that communicate through a shared data structure. Instead of using DMA hardware to implement the lower half, however, our remote disk driver will use a high-priority process that communicates over a network with a server by sending requests and receiving responses. Figure 18.1 illustrates the organization.

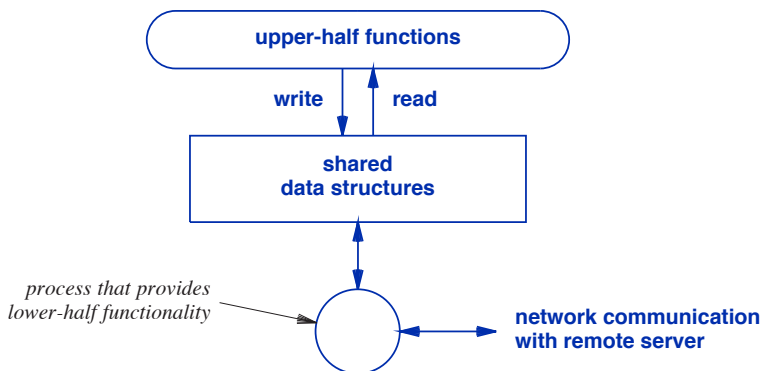


Figure 18.1 Organization of a remote disk driver.

Like a traditional disk driver, the shared data structure in our example contains two key items:

- A cache of recently accessed blocks
- A list of pending requests

A cache of recently accessed blocks. Because a disk operates much slower than a processor, a disk driver must be optimized to avoid making unnecessary transfers. The primary optimization technique consists of a *cache* kept in the shared data area. The cache stores recently accessed blocks, which allows subsequent operations on the same block to be performed on the copy in the cache. For example, if a user edits a document, stores a copy on disk, and then returns later to edit the same document, it is likely that the disk blocks containing the document will be in the cache, meaning that the document editing application can start much faster than if blocks had to be read from disk.

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.

18.6 Semantics Of Disk Operations

Although it can employ optimizations such as caching and delayed execution, 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:

$$op_1, op_2, op_3, \dots op_n$$

If op_i 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 queuing to enforce last-write semantics: the request list is a FIFO queue. That is:

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 discipline to insure that a process receives data according to last-write semantics.

18.7 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_BLKSIZE 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 8192 /* Stack size for comm. process */
#define RD_PRIO 200 /* Priority 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 rdbuf {
    struct rdbuf *rd_next;      /* ptr to next node on a list */
    struct rdbuf *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_BLKSIZE];  /* space to hold one disk block */
};

struct rdschlk {
    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 rdbuf *rd_rhnext;     /* head of request queue: next */
    struct rdbuf *rd_rhprev;     /* and previous */
    struct rdbuf *rd_rtnext;     /* tail of request queue: next */
    struct rdbuf *rd_rtprev;     /* (null) and previous */

    /* Cache head and tail */

    struct rdbuf *rd_chnext;     /* head of cache: next and */
    struct rdbuf *rd_chprev;     /* previous */
    struct rdbuf *rd_ctnext;     /* tail of cache: next (null) */
    struct rdbuf *rd_ctprev;     /* and previous */

    /* Free list head (singly-linked) */

    struct rdbuf *rd_free;       /* ptr to free list */

    pid32 rd_comproc;            /* process ID of comm. process */

```

```

    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) UDP port    */
    bool8    rd_registered;        /* has UDP port been registered?*/
};

extern struct rdschblk 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        /* wait two seconds for reply  */

/* 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_BLKSIZE];                   /* 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_BLKSIZE];                   /* 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()
```

18.8 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 *rdp;           /* ptr to device control 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 */

    rdp = &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 the high-priority process that communicates with the server, and it creates two semaphores that control processing. One semaphore, *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.

Another 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. Instead, some buffers must remain in the cache because processes are waiting to extract data. We will see how the cache and the semaphore are used later.

18.9 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 (rdp_ptr->rd_state != RD_FREE) {
    return SYSERR;
}
rdp_ptr->rd_state = RD_PEND;

/* Copy disk ID into free table slot */

idto = rdp_ptr->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),
                rdp_ptr );

/* Check response */

```

```

    if (retval == SYSERR) {
        return SYSERR;
    } else if (retval == TIMEOUT) {
        kprintf("Timeout during remote file open\n\r");
        return SYSERR;
    } else if (ntohs(resp.rd_status) != 0) {
        return SYSERR;
    }

    /* Change state of device to indicate currently open */

    rdptr->rd_state = RD_OPEN;

    /* Return device descriptor */

    return devptr->dvnum;
}

```

18.10 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,     /* buffer length            */
    int                 retries) /* number of retries        */

```

```

        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?  */

    /* For the first time after reboot, register the server port */

    if ( ! rdptr->rd_registered ) {
        retval = udp_register(0, rdptr->rd_ser_port,
                               rdptr->rd_loc_port);
        rdptr->rd_registered = TRUE;
    }

    if ( (int32)(localip = getlocalip()) == SYSERR ) {
        return SYSERR;
    }

    /* 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_send(rdptr->rd_ser_ip, rdptr->rd_ser_port,
                              localip, rdptr->rd_loc_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(0, rdptr->rd_ser_port,
                  rdptr->rd_loc_port, (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 obtains the local IP address (for use with UDP), assigns the next sequence number to the message, and enters a loop that iterates *RD_RETRIES* times. On each iteration, *rdscomm* calls *udp_send* 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 *type* of the reply matches the type of the request, that the sequence number of the reply matches the sequence number of the request that was sent, 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.

18.11 The Upper-Half Write Function (rdsWrite)

Because the remote disk system provides asynchronous *write* operations, the upper-half *write* function is easiest to understand. File *rdsWrite.c* contains the code:

```
/* 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 rdsblk *rdptr;      /* pointer to control block */
    struct rdbuff *bptr;       /* ptr 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_BLKSIZE);
        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_BLKSIZE);
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, but is not used.

18.12 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 wait 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, 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 con-

tains 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 allocated.

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 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;        /* ptr to buffer possibly on */
                                /* the request list */
    struct rdbuff *nptr;        /* ptr to "next" node on a */
                                /* list */
    struct rdbuff *pptr;        /* ptr to "previous" node on */
}
```

```

/* a list */
struct rdbuff *cptr; /* ptr used to walk 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) {
            return SYSERR;
        }
        memcpy(buff, bptr->rd_block, RD_BLKSIIZ);
        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_BLKSIIZ);
            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_BLKSIZE);
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 *reads*. Therefore, *rdsRead* must extract the old buffer from the cache and move it to the free list.

18.13 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 insures 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.

18.14 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,                /* a control function */
    int32 arg1,                /* argument #1 */
    int32 arg2                 /* argument #2 */
)
{
    struct rdschlk *rdpctr;     /* pointer to control block */
    struct rdbuff *bpctr;       /* ptr to buffer that will be
                                /* placed on the req. queue
    struct rdbuff *ppctr;       /* 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 */

    rdpctr = &rdstab[devptr->dvminor];
    if (rdpctr->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 rdbuf *)SYSEERR) {
    return SYSEERR;
}

/* 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 rdbuf *)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 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, 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.

18.15 Allocating A Disk Buffer (rdsbufalloc)

As we have seen, driver functions call *rdsbufalloc* when they need to allocate a buffer. 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 rdsblk *rdptr          /* ptr to device control block */
)
{
    struct rdbuff *bptr;          /* ptr to a buffer */
    struct rdbuff *pptr;          /* ptr to previous buffer */
    struct rdbuff *nptr;          /* ptr 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 rdbuf *)SYSERR;
}

```

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.

18.16 The Upper-Half Close Function (*rdsClose*)

A process invokes *close* to close the remote disk device and stop all communication. 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 rdsblk *rdptr;          /* ptr to control block entry */
    struct rdbuf *bptr;           /* ptr to buffer on a list */
    struct rdbuf *nptr;           /* ptr to next buff on the list */
    int32  nmoved;                /* number of buffers moved */

    /* Device must be open */

```

```
rdptr = &rdstab[devptr->dvmminor];
if (rdptr->rd_state != RD_OPEN) {
    return SYSERR;
}

/* Request queue must be empty */

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;
}
```

To close a remote disk device, all buffers must be moved back to the free list (re-creating 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.

18.17 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 that repeatedly extracts
 *              an item from the request queue and sends 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 */
}
```

[†]The synchronization code is found in file *rdsControl.c* on page 396.

```

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_BLKSIIZ; 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 */

        qpтр->rd_next = rdpтр->rd_free;
        rdpтр->rd_free = qpтр;
        signal(rdpтр->rd_availsem);
        break;
    }
    qpтр = qpтр->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;          /* rdscomm fills in an entry */
    idto = msg.rd_id;
    memset(idto, NULLCH, RD_IDLEN); /* initialize ID to zero */
    idfrom = rdpтр->rd_id;
    while ( (*idto++ = *idfrom++) != NULLCH ) { /* copy ID */
        ;
    }
    for (i=0; i<RD_BLKSIZE; i++) {
        msg.rd_data[i] = bptr->rd_block[i];
    }

    /* Unlink buffer from request queue */

    npтр = bptr->rd_next;
    pptr = bptr->rd_prev;
    pptr->rd_next = npтр;
    npтр->rd_prev = pptr;

    /* Insert buffer in the cache */

    pptr = (struct rdbuf *) &rdptr->rd_chnext;
    npтр = pptr->rd_next;
    bptr->rd_next = npтр;
    bptr->rd_prev = pptr;
    pptr->rd_next = bptr;
    npтр->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*.

18.18 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 insure that they occur in order. 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.

18.19 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.

ly 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, the 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 insure 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?

Operating System Design: The Xinu Approach, Linksys Version provides a comprehensive introduction to Operating System Design, using Xinu, a small, elegant operating system that serves as an example and a pattern for system design. The book focuses the discussion of operating systems on the microkernel operating system facilities used in embedded systems. Rather than introduce a new course to teach the important topics of embedded systems programming, this textbook takes the approach of integrating more embedded processing into existing operating systems courses. Designed for advanced undergraduate or graduate courses, the book prepares students for the increased demand for operating system expertise in industry.

Highlights

- Explains how each operating system abstraction can be built and shows how the abstractions can be organized into an elegant, efficient design
- Considers each level of the system individually, beginning with the raw hardware and ending with a working operating system
- Covers every part of the system, so a reader will see how an entire system fits together, not merely how one or two parts interact
- Provides source code for all pieces described in the text, leaving no mystery 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
- Demonstrates how each piece of an operating system fits into the design, in order to prepare the reader to understand alternative design choices

Beginning with the underlying machine and proceeding step by step through the design and implementation of an actual system, **Operating System Design: The Xinu Approach, Linksys Version** guides readers through the construction of a traditional process-based operating system using practical, straightforward primitives. It reviews the major system components and imposes a hierarchical design paradigm that organizes the components in an orderly and understandable manner.

Software and instructions for building a lab that allows students to experiment are available on the author's website: www.xinu.cs.purdue.edu



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

K13816

