

Robert Love

Third Edition



Linux Kernel Development

A thorough guide to the design and
implementation of the Linux kernel

Developer's Library



FREE SAMPLE CHAPTER

SHARE WITH OTHERS



Linux Kernel Development

Third Edition

Developer's Library

ESSENTIAL REFERENCES FOR PROGRAMMING PROFESSIONALS

Developer's Library books are designed to provide practicing programmers with unique, high-quality references and tutorials on the programming languages and technologies they use in their daily work.

All books in the *Developer's Library* are written by expert technology practitioners who are especially skilled at organizing and presenting information in a way that's useful for other programmers.

Key titles include some of the best, most widely acclaimed books within their topic areas:

PHP & MySQL Web Development

Luke Welling & Laura Thomson

ISBN 978-0-672-32916-6

MySQL

Paul DuBois

ISBN-13: 978-0-672-32938-8

Linux Kernel Development

Robert Love

ISBN-13: 978-0-672-32946-3

Python Essential Reference

David Beazley

ISBN-13: 978-0-672-32978-6

Programming in Objective-C 2.0

Stephen G. Kochan

ISBN-13: 978-0-321-56615-7

PostgreSQL

Korry Douglas

ISBN-13: 978-0-672-33015-5

Developer's Library books are available at most retail and online bookstores, as well as by subscription from Safari Books Online at **safari.informit.com**

**Developer's
Library**

informit.com/devlibrary

Linux Kernel Development

Third Edition

Robert Love

◆Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

Linux Kernel Development

Third Edition

Copyright © 2010 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise.

ISBN-13: 978-0-672-32946-3

ISBN-10: 0-672-32946-8

Library of Congress Cataloging-in-Publication Data:

Love, Robert.

Linux kernel development / Robert Love. — 3rd ed.

p. cm.

Includes bibliographical references and index.

ISBN 978-0-672-32946-3 (pbk. : alk. paper) 1. Linux. 2. Operating systems (Computers)
I. Title.

QA76.76.O63L674 2010

005.4'32—dc22

2010018961

Text printed in the United States on recycled paper at RR Donnelley, Crawfordsville, Indiana.
Third Printing: June 2011

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Acquisitions Editor
Mark Taber

Development
Editor
Michael Thurston

Technical Editor
Robert P. J. Day

Managing Editor
Sandra Schroeder

Senior Project
Editor
Tonya Simpson

Copy Editor
Apostrophe Editing
Services

Indexer
Brad Herriman

Proofreader
Debbie Williams

Publishing
Coordinator
Vanessa Evans

Book Designer
Gary Adair

Compositor
Mark Shirar



For Doris and Helen.



Contents at a Glance

1	Introduction to the Linux Kernel	1
2	Getting Started with the Kernel	11
3	Process Management	23
4	Process Scheduling	41
5	System Calls	69
6	Kernel Data Structures	85
7	Interrupts and Interrupt Handlers	113
8	Bottom Halves and Deferring Work	133
9	An Introduction to Kernel Synchronization	161
10	Kernel Synchronization Methods	175
11	Timers and Time Management	207
12	Memory Management	231
13	The Virtual Filesystem	261
14	The Block I/O Layer	289
15	The Process Address Space	305
16	The Page Cache and Page Writeback	323
17	Devices and Modules	337
18	Debugging	363
19	Portability	379
20	Patches, Hacking, and the Community	395
	Bibliography	407
	Index	411

Table of Contents

1 Introduction to the Linux Kernel 1

- History of Unix 1
- Along Came Linus: Introduction to Linux 3
- Overview of Operating Systems and Kernels 4
- Linux Versus Classic Unix Kernels 6
- Linux Kernel Versions 8
- The Linux Kernel Development Community 10
- Before We Begin 10

2 Getting Started with the Kernel 11

- Obtaining the Kernel Source 11
 - Using Git 11
 - Installing the Kernel Source 12
 - Using Patches 12
- The Kernel Source Tree 12
- Building the Kernel 13
 - Configuring the Kernel 14
 - Minimizing Build Noise 15
 - Spawning Multiple Build Jobs 16
 - Installing the New Kernel 16
- A Beast of a Different Nature 16
 - No libc or Standard Headers 17
 - GNU C 18
 - Inline Functions 18
 - Inline Assembly 19
 - Branch Annotation 19
 - No Memory Protection 20
 - No (Easy) Use of Floating Point 20
 - Small, Fixed-Size Stack 20
 - Synchronization and Concurrency 21
 - Importance of Portability 21
 - Conclusion 21

3 Process Management 23

- The Process 23
- Process Descriptor and the Task Structure 24
 - Allocating the Process Descriptor 25
 - Storing the Process Descriptor 26
 - Process State 27
 - Manipulating the Current Process State 29
 - Process Context 29
 - The Process Family Tree 29
- Process Creation 31
 - Copy-on-Write 31
 - Forking 32
 - `vfork()` 33
- The Linux Implementation of Threads 33
 - Creating Threads 34
 - Kernel Threads 35
- Process Termination 36
 - Removing the Process Descriptor 37
 - The Dilemma of the Parentless Task 38
 - Conclusion 40

4 Process Scheduling 41

- Multitasking 41
- Linux's Process Scheduler 42
- Policy 43
 - I/O-Bound Versus Processor-Bound Processes 43
 - Process Priority 44
 - Timeslice 45
 - The Scheduling Policy in Action 45
- The Linux Scheduling Algorithm 46
 - Scheduler Classes 46
 - Process Scheduling in Unix Systems 47
 - Fair Scheduling 48
- The Linux Scheduling Implementation 50
 - Time Accounting 50
 - The Scheduler Entity Structure 50
 - The Virtual Runtime 51

Process Selection	52
Picking the Next Task	53
Adding Processes to the Tree	54
Removing Processes from the Tree	56
The Scheduler Entry Point	57
Sleeping and Waking Up	58
Wait Queues	58
Waking Up	61
Preemption and Context Switching	62
User Preemption	62
Kernel Preemption	63
Real-Time Scheduling Policies	64
Scheduler-Related System Calls	65
Scheduling Policy and Priority-Related System Calls	66
Processor Affinity System Calls	66
Yielding Processor Time	66
Conclusion	67
5 System Calls	69
Communicating with the Kernel	69
APIs, POSIX, and the C Library	70
Syscalls	71
System Call Numbers	72
System Call Performance	72
System Call Handler	73
Denoting the Correct System Call	73
Parameter Passing	74
System Call Implementation	74
Implementing System Calls	74
Verifying the Parameters	75
System Call Context	78
Final Steps in Binding a System Call	79
Accessing the System Call from User-Space	81
Why Not to Implement a System Call	82
Conclusion	83

6 Kernel Data Structures 85

Linked Lists	85
Singly and Doubly Linked Lists	85
Circular Linked Lists	86
Moving Through a Linked List	87
The Linux Kernel's Implementation	88
The Linked List Structure	88
Defining a Linked List	89
List Heads	90
Manipulating Linked Lists	90
Adding a Node to a Linked List	90
Deleting a Node from a Linked List	91
Moving and Splicing Linked List Nodes	92
Traversing Linked Lists	93
The Basic Approach	93
The Usable Approach	93
Iterating Through a List Backward	94
Iterating While Removing	95
Other Linked List Methods	96
Queues	96
kfifo	97
Creating a Queue	97
Enqueuing Data	98
Dequeuing Data	98
Obtaining the Size of a Queue	98
Resetting and Destroying the Queue	99
Example Queue Usage	99
Maps	100
Initializing an idr	101
Allocating a New UID	101
Looking Up a UID	102
Removing a UID	103
Destroying an idr	103
Binary Trees	103
Binary Search Trees	104
Self-Balancing Binary Search Trees	105
Red-Black Trees	105
rbtrees	106

What Data Structure to Use, When	108
Algorithmic Complexity	109
Algorithms	109
Big-O Notation	109
Big Theta Notation	109
Time Complexity	110
Conclusion	111

7 Interrupts and Interrupt Handlers 113

Interrupts	113
Interrupt Handlers	114
Top Halves Versus Bottom Halves	115
Registering an Interrupt Handler	116
Interrupt Handler Flags	116
An Interrupt Example	117
Freeing an Interrupt Handler	118
Writing an Interrupt Handler	118
Shared Handlers	119
A Real-Life Interrupt Handler	120
Interrupt Context	122
Implementing Interrupt Handlers	123
/proc/interrupts	126
Interrupt Control	127
Disabling and Enabling Interrupts	127
Disabling a Specific Interrupt Line	129
Status of the Interrupt System	130
Conclusion	131

8 Bottom Halves and Deferring Work 133

Bottom Halves	134
Why Bottom Halves?	134
A World of Bottom Halves	135
The Original “Bottom Half”	135
Task Queues	135
Softirqs and Tasklets	136
Dispelling the Confusion	137

Softirqs	137
Implementing Softirqs	137
The Softirq Handler	138
Executing Softirqs	138
Using Softirqs	140
Assigning an Index	140
Registering Your Handler	141
Raising Your Softirq	141
Tasklets	142
Implementing Tasklets	142
The Tasklet Structure	142
Scheduling Tasklets	143
Using Tasklets	144
Declaring Your Tasklet	144
Writing Your Tasklet Handler	145
Scheduling Your Tasklet	145
ksoftirqd	146
The Old BH Mechanism	148
Work Queues	149
Implementing Work Queues	149
Data Structures Representing the Threads	149
Data Structures Representing the Work	150
Work Queue Implementation Summary	152
Using Work Queues	153
Creating Work	153
Your Work Queue Handler	153
Scheduling Work	153
Flushing Work	154
Creating New Work Queues	154
The Old Task Queue Mechanism	155
Which Bottom Half Should I Use?	156
Locking Between the Bottom Halves	157
Disabling Bottom Halves	157
Conclusion	159
9 An Introduction to Kernel Synchronization	161
Critical Regions and Race Conditions	162
Why Do We Need Protection?	162
The Single Variable	163

Locking	165
Causes of Concurrency	167
Knowing What to Protect	168
Deadlocks	169
Contention and Scalability	171
Conclusion	172

10 Kernel Synchronization Methods 175

Atomic Operations	175
Atomic Integer Operations	176
64-Bit Atomic Operations	180
Atomic Bitwise Operations	181
Spin Locks	183
Spin Lock Methods	184
Other Spin Lock Methods	186
Spin Locks and Bottom Halves	187
Reader-Writer Spin Locks	188
Semaphores	190
Counting and Binary Semaphores	191
Creating and Initializing Semaphores	192
Using Semaphores	193
Reader-Writer Semaphores	194
Mutexes	195
Semaphores Versus Mutexes	197
Spin Locks Versus Mutexes	197
Completion Variables	197
BKL: The Big Kernel Lock	198
Sequential Locks	200
Preemption Disabling	201
Ordering and Barriers	203
Conclusion	206

11 Timers and Time Management 207

Kernel Notion of Time	208
The Tick Rate: HZ	208
The Ideal HZ Value	210
Advantages with a Larger HZ	210
Disadvantages with a Larger HZ	211

Jiffies	212
Internal Representation of Jiffies	213
Jiffies Wraparound	214
User-Space and HZ	216
Hardware Clocks and Timers	216
Real-Time Clock	217
System Timer	217
The Timer Interrupt Handler	217
The Time of Day	220
Timers	222
Using Timers	222
Timer Race Conditions	224
Timer Implementation	224
Delaying Execution	225
Busy Looping	225
Small Delays	226
schedule_timeout()	227
schedule_timeout() Implementation	228
Sleeping on a Wait Queue, with a Timeout	229
Conclusion	230

12 Memory Management 231

Pages	231
Zones	233
Getting Pages	235
Getting Zeroed Pages	236
Freeing Pages	237
kmalloc()	238
gfp_mask Flags	238
Action Modifiers	239
Zone Modifiers	240
Type Flags	241
kfree()	243
vmalloc()	244
Slab Layer	245
Design of the Slab Layer	246

Slab Allocator Interface	249
Allocating from the Cache	250
Example of Using the Slab Allocator	251
Statically Allocating on the Stack	252
Single-Page Kernel Stacks	252
Playing Fair on the Stack	253
High Memory Mappings	253
Permanent Mappings	254
Temporary Mappings	254
Per-CPU Allocations	255
The New percpu Interface	256
Per-CPU Data at Compile-Time	256
Per-CPU Data at Runtime	257
Reasons for Using Per-CPU Data	258
Picking an Allocation Method	259
Conclusion	260

13 The Virtual Filesystem 261

Common Filesystem Interface	261
Filesystem Abstraction Layer	262
Unix Filesystems	263
VFS Objects and Their Data Structures	265
The Superblock Object	266
Superblock Operations	267
The Inode Object	270
Inode Operations	271
The Dentry Object	275
Dentry State	276
The Dentry Cache	276
Dentry Operations	278
The File Object	279
File Operations	280
Data Structures Associated with Filesystems	285
Data Structures Associated with a Process	286
Conclusion	288

14 The Block I/O Layer 289

- Anatomy of a Block Device 290
- Buffers and Buffer Heads 291
- The bio Structure 294
 - I/O vectors 295
 - The Old Versus the New 296
- Request Queues 297
- I/O Schedulers 297
 - The Job of an I/O Scheduler 298
 - The Linus Elevator 299
 - The Deadline I/O Scheduler 300
 - The Anticipatory I/O Scheduler 302
 - The Complete Fair Queuing I/O Scheduler 303
 - The Noop I/O Scheduler 303
 - I/O Scheduler Selection 304
 - Conclusion 304

15 The Process Address Space 305

- Address Spaces 305
- The Memory Descriptor 306
 - Allocating a Memory Descriptor 308
 - Destroying a Memory Descriptor 309
 - The mm_struct and Kernel Threads 309
- Virtual Memory Areas 309
 - VMA Flags 311
 - VMA Operations 312
 - Lists and Trees of Memory Areas 313
 - Memory Areas in Real Life 314
- Manipulating Memory Areas 315
 - find_vma() 316
 - find_vma_prev() 317
 - find_vma_intersection() 317
- mmap() and do_mmap(): Creating an Address Interval 318
- munmap() and do_munmap(): Removing an Address Interval 320
- Page Tables 320
- Conclusion 322

16 The Page Cache and Page Writeback 323

- Approaches to Caching 323
 - Write Caching 324
 - Cache Eviction 324
 - Least Recently Used 325
 - The Two-List Strategy 325
- The Linux Page Cache 326
 - The `address_space` Object 326
 - `address_space` Operations 328
 - Radix Tree 330
 - The Old Page Hash Table 330
- The Buffer Cache 330
- The Flusher Threads 331
 - Laptop Mode 333
 - History: `bdflush`, `kupdated`, and `pdflush` 333
 - Avoiding Congestion with Multiple Threads 334
- Conclusion 335

17 Devices and Modules 337

- Device Types 337
- Modules 338
 - Hello, World! 338
 - Building Modules 340
 - Living in the Source Tree 340
 - Living Externally 342
 - Installing Modules 342
 - Generating Module Dependencies 342
 - Loading Modules 343
 - Managing Configuration Options 344
 - Module Parameters 346
 - Exported Symbols 348
- The Device Model 348
 - Kobjects 349
 - Ktypes 350
 - Ksets 351
 - Interrelation of Kobjects, Ktypes, and Ksets 351
 - Managing and Manipulating Kobjects 352

Reference Counts	353
Incrementing and Decrementing Reference Counts	354
Krefs	354
sysfs	355
Adding and Removing kobjects from sysfs	357
Adding Files to sysfs	358
Default Attributes	358
Creating New Attributes	359
Destroying Attributes	360
sysfs Conventions	360
The Kernel Events Layer	361
Conclusion	362
18 Debugging	363
Getting Started	363
Bugs in the Kernel	364
Debugging by Printing	364
Robustness	365
Loglevels	365
The Log Buffer	366
syslogd and klogd	367
Transposing printf() and printk()	367
Oops	367
ksymoops	369
kallsyms	369
Kernel Debugging Options	370
Asserting Bugs and Dumping Information	370
Magic SysRq Key	371
The Saga of a Kernel Debugger	372
gdb	372
kgdb	373
Poking and Probing the System	373
Using UID as a Conditional	373
Using Condition Variables	374
Using Statistics	374
Rate and Occurrence Limiting Your Debugging	375

Binary Searching to Find the Culprit Change	376
Binary Searching with Git	376
When All Else Fails: The Community	377
Conclusion	378

19 Portability 379

Portable Operating Systems	379
History of Portability in Linux	380
Word Size and Data Types	381
Opaque Types	384
Special Types	384
Explicitly Sized Types	385
Signedness of Chars	386
Data Alignment	386
Avoiding Alignment Issues	387
Alignment of Nonstandard Types	387
Structure Padding	387
Byte Order	389
Time	391
Page Size	391
Processor Ordering	392
SMP, Kernel Preemption, and High Memory	393
Conclusion	393

20 Patches, Hacking, and the Community 395

The Community	395
Linux Coding Style	396
Indention	396
Switch Statements	396
Spacing	397
Braces	398
Line Length	399
Naming	400
Functions	400
Comments	400
Typedefs	401
Use Existing Routines	402

Minimize <code>ifdefs</code> in the Source	402
Structure Initializers	402
Fixing Up Code Ex Post Facto	403
Chain of Command	403
Submitting Bug Reports	403
Patches	404
Generating Patches	404
Generating Patches with Git	405
Submitting Patches	406
Conclusion	406

Bibliography 407

Index 411

Foreword

As the Linux kernel and the applications that use it become more widely used, we are seeing an increasing number of system software developers who wish to become involved in the development and maintenance of Linux. Some of these engineers are motivated purely by personal interest, some work for Linux companies, some work for hardware manufacturers, and some are involved with in-house development projects.

But all face a common problem: The learning curve for the kernel is getting longer and steeper. The system is becoming increasingly complex, and it is very large. And as the years pass, the current members of the kernel development team gain deeper and broader knowledge of the kernel's internals, which widens the gap between them and newcomers.

I believe that this declining accessibility of the Linux source base is already a problem for the quality of the kernel, and it will become more serious over time. Those who care for Linux clearly have an interest in increasing the number of developers who can contribute to the kernel.

One approach to this problem is to keep the code clean: sensible interfaces, consistent layout, "do one thing, do it well," and so on. This is Linus Torvalds' solution.

The approach that I counsel is to liberally apply commentary to the code: words that the reader can use to understand what the coder intended to achieve at the time. (The process of identifying divergences between the intent and the implementation is known as debugging. It is hard to do this if the intent is not known.)

But even code commentary does not provide the broad-sweep view of what a major subsystem is intended to do, and of how its developers set about doing it. This, the starting point of understanding, is what the written word serves best.

Robert Love's contribution provides a means by which experienced developers can gain that essential view of what services the kernel subsystems are supposed to provide, and of how they set about providing them. This will be sufficient knowledge for many people: the curious, the application developers, those who wish to evaluate the kernel's design, and others.

But the book is also a stepping stone to take aspiring kernel developers to the next stage, which is making alterations to the kernel to achieve some defined objective. I would encourage aspiring developers to get their hands dirty: The best way to understand a part of the kernel is to make changes to it. Making a change forces the developer to a level of understanding which merely reading the code does not provide. The serious kernel developer will join the development mailing lists and will interact with other developers. This interaction is the primary means by which kernel contributors learn

and stay abreast. Robert covers the mechanics and culture of this important part of kernel life well.

Please enjoy and learn from Robert's book. And should you decide to take the next step and become a member of the kernel development community, consider yourself welcomed in advance. We value and measure people by the usefulness of their contributions, and when you contribute to Linux, you do so in the knowledge that your work is of small but immediate benefit to tens or even hundreds of millions of human beings. This is a most enjoyable privilege and responsibility.

Andrew Morton

Preface

When I was first approached about converting my experiences with the Linux kernel into a book, I proceeded with trepidation. What would place my book at the top of its subject? I was not interested unless I could do something special, a best-in-class work.

I realized that I could offer a unique approach to the topic. My job is hacking the kernel. My hobby is hacking the kernel. My love is hacking the kernel. Over the years, I have accumulated interesting anecdotes and insider tips. With my experiences, I could write a book on how to hack the kernel and—just as important—how *not* to hack the kernel. First and foremost, this is a book about the design and implementation of the Linux kernel. This book's approach differs from would-be competitors, however, in that the information is given with a slant to learning enough to actually get work done—and getting it done right. I am a pragmatic engineer and this is a practical book. It should be fun, easy to read, and useful.

I hope that readers can walk away from this work with a better understanding of the rules (written and unwritten) of the Linux kernel. I intend that you, fresh from reading this book and the kernel source code, can jump in and start writing useful, correct, clean kernel code. Of course, you can read this book just for fun, too.

That was the first edition. Time has passed, and now we return once more to the fray. This third edition offers quite a bit over the first and second: intense polish and revision, updates, and many fresh sections and all new chapters. This edition incorporates changes in the kernel since the second edition. More important, however, is the decision made by the Linux kernel community to not proceed with a 2.7 development kernel in the near to mid-term.¹ Instead, kernel developers plan to continue developing and stabilizing the 2.6 series. This decision has many implications, but the item of relevance to this book is that there is quite a bit of staying power in a contemporary book on the 2.6 Linux kernel. As the Linux kernel matures, there is a greater chance of a snapshot of the kernel remaining representative long into the future. This book functions as the canonical documentation for the kernel, documenting it with both an understanding of its history and an eye to the future.

Using This Book

Developing code in the kernel does not require genius, magic, or a bushy Unix-hacker beard. The kernel, although having some interesting rules of its own, is not much different from any other large software endeavor. You need to master many details—as with any big project—but the differences are quantitative, not qualitative.

¹ This decision was made in the summer of 2004 at the annual Linux Kernel Developers Summit in Ottawa, Canada. Your author was an invited attendee.

It is imperative that you utilize the source. The open availability of the source code for the Linux system is a rare gift that you must not take for granted. It is not sufficient *only* to read the source, however. You need to dig in and change some code. Find a bug and fix it. Improve the drivers for your hardware. Add some new functionality, even if it is trivial. Find an itch and scratch it! Only when you *write* code will it all come together.

Kernel Version

This book is based on the 2.6 Linux kernel series. It does not cover older kernels, except for historical relevance. We discuss, for example, how certain subsystems are implemented in the 2.4 Linux kernel series, as their simpler implementations are helpful teaching aids. Specifically, this book is up to date as of Linux kernel version 2.6.34. Although the kernel is a moving target and no effort can hope to capture such a dynamic beast in a timeless manner, my intention is that this book is relevant for developers and users of both older and newer kernels.

Although this book discusses the 2.6.34 kernel, I have made an effort to ensure the material is factually correct with respect to the 2.6.32 kernel as well. That latter version is sanctioned as the “enterprise” kernel by the various Linux distributions, ensuring we will continue to see it in production systems and under active development for many years. (2.6.9, 2.6.18, and 2.6.27 were similar “long-term” releases.)

Audience

This book targets Linux developers and users who are interested in understanding the Linux kernel. It is *not* a line-by-line commentary of the kernel source. Nor is it a guide to developing drivers or a reference on the kernel API. Instead, the goal of this book is to provide enough information on the design and implementation of the Linux kernel that a sufficiently accomplished programmer can begin developing code in the kernel. Kernel development can be fun and rewarding, and I want to introduce the reader to that world as readily as possible. This book, however, in discussing both theory and application, should appeal to readers of both academic and practical persuasions. I have always been of the mind that one needs to understand the theory to understand the application, but I try to balance the two in this work. I hope that whatever your motivations for understanding the Linux kernel, this book explains the design and implementation sufficiently for your needs.

Thus, this book covers both the usage of core kernel systems and their design and implementation. I think this is important and deserves a moment’s discussion. A good example is Chapter 8, “Bottom Halves and Deferring Work,” which covers a component of device drivers called bottom halves. In that chapter, I discuss both the design and implementation of the kernel’s bottom-half mechanisms (which a core kernel developer or academic might find interesting) and how to actually use the exported interfaces to implement your own bottom half (which a device driver developer or casual hacker can find pertinent). I believe all groups can find both discussions relevant. The core kernel

developer, who certainly needs to understand the inner workings of the kernel, should have a good understanding of how the interfaces are actually used. At the same time, a device driver writer can benefit from a good understanding of the implementation behind the interface.

This is akin to learning some library's API versus studying the actual implementation of the library. At first glance, an application programmer needs to understand only the API—it is often taught to treat interfaces as a black box. Likewise, a library developer is concerned only with the library's design and implementation. I believe, however, both parties should invest time in learning the other half. An application programmer who better understands the underlying operating system can make much greater use of it. Similarly, the library developer should not grow out of touch with the reality and practicality of the applications that use the library. Consequently, I discuss both the design and usage of kernel subsystems, not only in hopes that this book will be useful to either party, but also in hopes that the *whole* book is useful to both parties.

I assume that the reader knows the C programming language and is familiar with Linux systems. Some experience with operating system design and related computer science topics is beneficial, but I try to explain concepts as much as possible—if not, the Bibliography includes some excellent books on operating system design.

This book is appropriate for an undergraduate course introducing operating system design as the *applied* text if accompanied by an introductory book on theory. This book should fare well either in an advanced undergraduate course or in a graduate-level course without ancillary material.

Third Edition Acknowledgments

Like most authors, I did not write this book in a cave, which is a good thing, because there are bears in caves. Consequently many hearts and minds contributed to the completion of this manuscript. Although no list could be complete, it is my sincere pleasure to acknowledge the assistance of many friends and colleagues who provided encouragement, knowledge, and constructive criticism.

First, I would like to thank my team at Addison–Wesley and Pearson who worked long and hard to make this a better book, particularly Mark Taber for spearheading this third edition from conception to final product; Michael Thurston, development editor; and Tonya Simpson, project editor.

A special thanks to my technical editor on this edition, Robert P.J. Day. His insight, experience, and corrections improved this book immeasurably. Despite his sterling effort, however, any remaining mistakes remain my own. I have the same gratitude to Adam Belay, Zack Brown, Martin Pool, and Chris Rivera, whose excellent technical editing efforts on the first and second editions still shine through.

Many fellow kernel developers answered questions, provided support, or simply wrote code interesting enough on which to write a book. They include Andrea Arcangeli, Alan Cox, Greg Kroah-Hartman, Dave Miller, Patrick Mochel, Andrew Morton, Nick Piggin, and Linus Torvalds.

A big thank you to my colleagues at Google, the most creative and intelligent group with which I have ever had the pleasure to work. Too many names would fill these pages if I listed them all, but I will single out Alan Blount, Jay Crim, Chris Danis, Chris DiBona, Eric Flatt, Mike Lockwood, San Mehat, Brian Rogan, Brian Swetland, Jon Trowbridge, and Steve Vinter for their friendship, knowledge, and support.

Respect and love to Paul Amici, Mikey Babbitt, Keith Barbag, Jacob Berkman, Nat Friedman, Dustin Hall, Joyce Hawkins, Miguel de Icaza, Jimmy Krehl, Doris Love, Linda Love, Brette Luck, Randy O'Dowd, Sal Ribaud and mother, Chris Rivera, Carolyn Rodon, Joey Shaw, Sarah Stewart, Jeremy VanDoren and family, Luis Villa, Steve Weisberg and family, and Helen Whisnant.

Finally, thank you to my parents for so much, particularly my well-proportioned ears. Happy Hacking!

Robert Love
Boston

About the Author

Robert Love is an open source programmer, speaker, and author who has been using and contributing to Linux for more than 15 years. Robert is currently senior software engineer at Google, where he was a member of the team that developed the Android mobile platform's kernel. Prior to Google, he was Chief Architect, Linux Desktop, at Novell. Before Novell, he was a kernel engineer at MontaVista Software and Ximian.

Robert's kernel projects include the preemptive kernel, the process scheduler, the kernel events layer, inotify, VM enhancements, and several device drivers.

Robert has given numerous talks on and has written multiple articles about the Linux kernel. He is a contributing editor for *Linux Journal*. His other books include *Linux System Programming* and *Linux in a Nutshell*.

Robert received a B.A. degree in mathematics and a B.S. degree in computer science from the University of Florida. He lives in Boston.

Getting Started with the Kernel

In this chapter, we introduce some of the basics of the Linux kernel: where to get its source, how to compile it, and how to install the new kernel. We then go over the differences between the kernel and user-space programs and common programming constructs used in the kernel. Although the kernel certainly is unique in many ways, at the end of the day it is little different from any other large software project.

Obtaining the Kernel Source

The current Linux source code is always available in both a complete *tarball* (an archive created with the *tar* command) and an incremental patch from the official home of the Linux kernel, <http://www.kernel.org>.

Unless you have a specific reason to work with an older version of the Linux source, you *always* want the latest code. The repository at [kernel.org](http://www.kernel.org) is the place to get it, along with additional patches from a number of leading kernel developers.

Using Git

Over the last couple of years, the kernel hackers, led by Linus himself, have begun using a new version control system to manage the Linux kernel source. Linus created this system, called *Git*, with speed in mind. Unlike traditional systems such as *CVS*, *Git* is distributed, and its usage and workflow is consequently unfamiliar to many developers. I strongly recommend using *Git* to download and manage the Linux kernel source.

You can use *Git* to obtain a copy of the latest “pushed” version of Linus’s tree:

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git
```

When checked out, you can update your tree to Linus’s latest:

```
$ git pull
```

With these two commands, you can obtain and subsequently keep up to date with the official kernel tree. To commit and manage your own changes, see Chapter 20, “Patches,

Hacking, and the Community.” A complete discussion of Git is outside the scope of this book; many online resources provide excellent guides.

Installing the Kernel Source

The kernel tarball is distributed in both GNU zip (*gzip*) and *bzip2* format. *Bzip2* is the default and preferred format because it generally compresses quite a bit better than *gzip*. The Linux kernel tarball in *bzip2* format is named `linux-x.y.z.tar.bz2`, where *x.y.z* is the version of that particular release of the kernel source. After downloading the source, uncompressing and untarring it is simple. If your tarball is compressed with *bzip2*, run

```
$ tar xvjf linux-x.y.z.tar.bz2
```

If it is compressed with GNU zip, run

```
$ tar xvzf linux-x.y.z.tar.gz
```

This uncompresses and untars the source to the directory `linux-x.y.z`. If you use *git* to obtain and manage the kernel source, you do not need to download the tarball. Just run the *git clone* command as described and *git* downloads and unpacks the latest source.

Where to Install and Hack on the Source

The kernel source is typically installed in `/usr/src/linux`. You should not use this source tree for development because the kernel version against which your C library is compiled is often linked to this tree. Moreover, you should not require root in order to make changes to the kernel—instead, work out of your home directory and use root only to install new kernels. Even when installing a new kernel, `/usr/src/linux` should remain untouched.

Using Patches

Throughout the Linux kernel community, patches are the *lingua franca* of communication. You will distribute your code changes in patches and receive code from others as patches. *Incremental patches* provide an easy way to move from one kernel tree to the next. Instead of downloading each large tarball of the kernel source, you can simply apply an incremental patch to go from one version to the next. This saves everyone bandwidth and you time. To apply an incremental patch, from *inside* your kernel source tree, simply run

```
$ patch -p1 < ../patch-x.y.z
```

Generally, a patch to a given version of the kernel is applied against the previous version. Generating and applying patches is discussed in much more depth in later chapters.

The Kernel Source Tree

The kernel source tree is divided into a number of directories, most of which contain many more subdirectories. The directories in the root of the source tree, along with their descriptions, are listed in Table 2.1.

Table 2.1 Directories in the Root of the Kernel Source Tree

Directory	Description
<code>arch</code>	Architecture-specific source
<code>block</code>	Block I/O layer
<code>crypto</code>	Crypto API
<code>Documentation</code>	Kernel source documentation
<code>drivers</code>	Device drivers
<code>firmware</code>	Device firmware needed to use certain drivers
<code>fs</code>	The VFS and the individual filesystems
<code>include</code>	Kernel headers
<code>init</code>	Kernel boot and initialization
<code>ipc</code>	Interprocess communication code
<code>kernel</code>	Core subsystems, such as the scheduler
<code>lib</code>	Helper routines
<code>mm</code>	Memory management subsystem and the VM
<code>net</code>	Networking subsystem
<code>samples</code>	Sample, demonstrative code
<code>scripts</code>	Scripts used to build the kernel
<code>security</code>	Linux Security Module
<code>sound</code>	Sound subsystem
<code>usr</code>	Early user-space code (called <code>initramfs</code>)
<code>tools</code>	Tools helpful for developing Linux
<code>virt</code>	Virtualization infrastructure

A number of files in the root of the source tree deserve mention. The file `COPYING` is the kernel license (the GNU GPL v2). `CREDITS` is a listing of developers with more than a trivial amount of code in the kernel. `MAINTAINERS` lists the names of the individuals who maintain subsystems and drivers in the kernel. `Makefile` is the base kernel Makefile.

Building the Kernel

Building the kernel is easy. It is surprisingly easier than compiling and installing other system-level components, such as `glibc`. The 2.6 kernel series introduced a new configuration and build system, which made the job even easier and is a welcome improvement over earlier releases.

Configuring the Kernel

Because the Linux source code is available, it follows that you can configure and custom tailor it before compiling. Indeed, it is possible to compile support into your kernel for only the specific features and drivers you want. Configuring the kernel is a required step before building it. Because the kernel offers myriad features and supports a varied basket of hardware, there is a *lot* to configure. Kernel configuration is controlled by configuration options, which are prefixed by `CONFIG` in the form `CONFIG_FEATURE`. For example, symmetrical multiprocessing (SMP) is controlled by the configuration option `CONFIG_SMP`. If this option is set, SMP is enabled; if unset, SMP is disabled. The configure options are used both to decide which files to build and to manipulate code via preprocessor directives.

Configuration options that control the build process are either *Booleans* or *tristates*. A Boolean option is either *yes* or *no*. Kernel features, such as `CONFIG_PREEMPT`, are usually Booleans. A tristate option is one of *yes*, *no*, or *module*. The *module* setting represents a configuration option that is set but is to be compiled as a module (that is, a separate dynamically loadable object). In the case of tristates, a *yes* option explicitly means to compile the code into the main kernel image and not as a module. Drivers are usually represented by tristates.

Configuration options can also be strings or integers. These options do not control the build process but instead specify values that kernel source can access as a preprocessor macro. For example, a configuration option can specify the size of a statically allocated array.

Vendor kernels, such as those provided by Canonical for Ubuntu or Red Hat for Fedora, are precompiled as part of the distribution. Such kernels typically enable a good cross section of the needed kernel features and compile nearly all the drivers as modules. This provides for a great base kernel with support for a wide range of hardware as separate modules. For better or worse, as a kernel hacker, you need to compile your own kernels and learn what modules to include on your own.

Thankfully, the kernel provides multiple tools to facilitate configuration. The simplest tool is a text-based command-line utility:

```
$ make config
```

This utility goes through each option, one by one, and asks the user to interactively select *yes*, *no*, or (for tristates) *module*. Because this takes a *long* time, unless you are paid by the hour, you should use an ncurses-based graphical utility:

```
$ make menuconfig
```

Or a gtk+-based graphical utility:

```
$ make gconfig
```

These three utilities divide the various configuration options into categories, such as “Processor Type and Features.” You can move through the categories, view the kernel options, and of course change their values.

This command creates a configuration based on the defaults for your architecture:

```
$ make defconfig
```

Although these defaults are somewhat arbitrary (on i386, they are rumored to be Linus's configuration!), they provide a good start if you have never configured the kernel. To get off and running quickly, run this command and then go back and ensure that configuration options for your hardware are enabled.

The configuration options are stored in the root of the kernel source tree in a file named `.config`. You may find it easier (as most of the kernel developers do) to just edit this file directly. It is quite easy to search for and change the value of the configuration options. After making changes to your configuration file, or when using an existing configuration file on a new kernel tree, you can validate and update the configuration:

```
$ make oldconfig
```

You should always run this before building a kernel.

The configuration option `CONFIG_IKCONFIG_PROC` places the complete kernel configuration file, compressed, at `/proc/config.gz`. This makes it easy to clone your current configuration when building a new kernel. If your current kernel has this option enabled, you can copy the configuration out of `/proc` and use it to build a new kernel:

```
$ zcat /proc/config.gz > .config
$ make oldconfig
```

After the kernel configuration is set—however you do it—you can build it with a single command:

```
$ make
```

Unlike kernels before 2.6, you no longer need to run `make dep` before building the kernel—the dependency tree is maintained automatically. You also do not need to specify a specific build type, such as `bzImage`, or build modules separately, as you did in old versions. The default Makefile rule will handle everything.

Minimizing Build Noise

A trick to minimize build noise, but still see warnings and errors, is to redirect the output from `make`:

```
$ make > ../detritus
```

If you need to see the build output, you can read the file. Because the warnings and errors are output to standard error, however, you normally do not need to. In fact, I just do

```
$ make > /dev/null
```

This redirects all the worthless output to that big, ominous sink of no return, `/dev/null`.

Spawning Multiple Build Jobs

The `make` program provides a feature to split the build process into a number of parallel *jobs*. Each of these jobs then runs separately and concurrently, significantly speeding up the build process on multiprocessing systems. It also improves processor utilization because the time to build a large source tree includes significant time in I/O wait (time in which the process is idle waiting for an I/O request to complete).

By default, `make` spawns only a single job because Makefiles all too often have incorrect dependency information. With incorrect dependencies, multiple jobs can step on each other's toes, resulting in errors in the build process. The kernel's Makefiles have correct dependency information, so spawning multiple jobs does not result in failures. To build the kernel with multiple `make` jobs, use

```
$ make -jn
```

Here, *n* is the number of jobs to spawn. Usual practice is to spawn one or two jobs per processor. For example, on a 16-core machine, you might do

```
$ make -j32 > /dev/null
```

Using utilities such as the excellent `distcc` or `ccache` can also dramatically improve kernel build time.

Installing the New Kernel

After the kernel is built, you need to install it. How it is installed is architecture- and boot loader-dependent—consult the directions for your boot loader on where to copy the kernel image and how to set it up to boot. Always keep a known-safe kernel or two around in case your new kernel has problems!

As an example, on an x86 system using `grub`, you would copy `arch/i386/boot/bzImage` to `/boot`, name it something like `vmlinuz-version`, and edit `/boot/grub/grub.conf`, adding a new entry for the new kernel. Systems using `LILO` to boot would instead edit `/etc/lilo.conf` and then rerun `lilo`.

Installing modules, thankfully, is automated and architecture-independent. As root, simply run

```
% make modules_install
```

This installs all the compiled modules to their correct home under `/lib/modules`.

The build process also creates the file `System.map` in the root of the kernel source tree. It contains a symbol lookup table, mapping kernel symbols to their start addresses. This is used during debugging to translate memory addresses to function and variable names.

A Beast of a Different Nature

The Linux kernel has several unique attributes as compared to a normal user-space application. Although these differences do not necessarily make developing kernel code *harder* than developing user-space code, they certainly make doing so *different*.

These characteristics make the kernel a beast of a different nature. Some of the usual rules are bent; other rules are entirely new. Although some of the differences are obvious (we all know the kernel can do anything it wants), others are not so obvious. The most important of these differences are

- The kernel has access to neither the C library nor the standard C headers.
- The kernel is coded in GNU C.
- The kernel lacks the memory protection afforded to user-space.
- The kernel cannot easily execute floating-point operations.
- The kernel has a small per-process fixed-size stack.
- Because the kernel has asynchronous interrupts, is preemptive, and supports SMP, synchronization and concurrency are major concerns within the kernel.
- Portability is important.

Let's briefly look at each of these issues because all kernel developers must keep them in mind.

No libc or Standard Headers

Unlike a user-space application, the kernel is not linked against the standard C library—or any other library, for that matter. There are multiple reasons for this, including a chicken-and-the-egg situation, but the primary reason is speed and size. The full C library—or even a decent subset of it—is too large and too inefficient for the kernel.

Do not fret: Many of the usual libc functions are implemented inside the kernel. For example, the common string manipulation functions are in `lib/string.c`. Just include the header file `<linux/string.h>` and have at them.

Header Files

When I talk about header files in this book, I am referring to the kernel header files that are part of the kernel source tree. Kernel source files cannot include outside headers, just as they cannot use outside libraries.

The base files are located in the `include/` directory in the root of the kernel source tree. For example, the header file `<linux/inotify.h>` is located at `include/linux/inotify.h` in the kernel source tree.

A set of architecture-specific header files are located in `arch/<architecture>/include/asm` in the kernel source tree. For example, if compiling for the x86 architecture, your architecture-specific headers are in `arch/x86/include/asm`. Source code includes these headers via just the `asm/` prefix, for example `<asm/ioctl.h>`.

Of the missing functions, the most familiar is `printf()`. The kernel does not have access to `printf()`, but it does provide `printk()`, which works pretty much the same as its more familiar cousin. The `printk()` function copies the formatted string into the kernel log buffer, which is normally read by the `syslog` program. Usage is similar to `printf()`:

```
printk("Hello world! A string '%s' and an integer '%d'\n", str, i);
```

One notable difference between `printf()` and `printk()` is that `printk()` enables you to specify a priority flag. This flag is used by `syslogd` to decide where to display kernel messages. Here is an example of these priorities:

```
printk(KERN_ERR "this is an error!\n");
```

Note there is no comma between `KERN_ERR` and the printed message. This is intentional; the priority flag is a preprocessor-define representing a string literal, which is concatenated onto the printed message during compilation. We use `printk()` throughout this book.

GNU C

Like any self-respecting Unix kernel, the Linux kernel is programmed in C. Perhaps surprisingly, the kernel is not programmed in strict ANSI C. Instead, where applicable, the kernel developers make use of various language extensions available in `gcc` (the GNU Compiler Collection, which contains the C compiler used to compile the kernel and most everything else written in C on a Linux system).

The kernel developers use both ISO C99¹ and GNU C extensions to the C language. These changes wed the Linux kernel to `gcc`, although recently one other compiler, the Intel C compiler, has sufficiently supported enough `gcc` features that it, too, can compile the Linux kernel. The earliest supported `gcc` version is 3.2; `gcc` version 4.4 or later is recommended. The ISO C99 extensions that the kernel uses are nothing special and, because C99 is an official revision of the C language, are slowly cropping up in a lot of other code. The more unfamiliar deviations from standard ANSI C are those provided by GNU C. Let's look at some of the more interesting extensions that you will see in the kernel; these changes differentiate kernel code from other projects with which you might be familiar.

Inline Functions

Both C99 and GNU C support *inline functions*. An inline function is, as its name suggests, inserted inline into each function call site. This eliminates the overhead of function invocation and return (register saving and restore) and allows for potentially greater optimization as the compiler can optimize both the caller and the called function as one. As a downside (nothing in life is free), code size increases because the contents of the function are copied into all the callers, which increases memory consumption and instruction cache footprint. Kernel developers use inline functions for small time-critical functions.

¹ ISO C99 is the latest major revision to the ISO C standard. C99 adds numerous enhancements to the previous major revision, ISO C90, including designated initializers, variable length arrays, C++-style comments, and the *long long* and *complex* types. The Linux kernel, however, employs only a subset of C99 features.

Making large functions inline, especially those used more than once or that are not exceedingly time critical, is frowned upon.

An inline function is declared when the keywords `static` and `inline` are used as part of the function definition. For example

```
static inline void wolf(unsigned long tail_size)
```

The function declaration must precede any usage, or else the compiler cannot make the function inline. Common practice is to place inline functions in header files. Because they are marked `static`, an exported function is not created. If an inline function is used by only one file, it can instead be placed toward the top of just that file.

In the kernel, using inline functions is preferred over complicated macros for reasons of type safety and readability.

Inline Assembly

The gcc C compiler enables the embedding of assembly instructions in otherwise normal C functions. This feature, of course, is used in only those parts of the kernel that are unique to a given system architecture.

The `asm()` compiler directive is used to inline assembly code. For example, this inline assembly directive executes the x86 processor's `rdtsc` instruction, which returns the value of the timestamp (`tsc`) register:

```
unsigned int low, high;
asm volatile("rdtsc" : "=a" (low), "=d" (high));
/* low and high now contain the lower and upper 32-bits of the 64-bit tsc */
```

The Linux kernel is written in a mixture of C and assembly, with assembly relegated to low-level architecture and fast path code. The vast majority of kernel code is programmed in straight C.

Branch Annotation

The gcc C compiler has a built-in directive that optimizes conditional branches as either very likely taken or very unlikely taken. The compiler uses the directive to appropriately optimize the branch. The kernel wraps the directive in easy-to-use macros, `likely()` and `unlikely()`.

For example, consider an `if` statement such as the following:

```
if (error) {
    /* ... */
}
```

To mark this branch as very unlikely taken (that is, likely not taken):

```
/* we predict 'error' is nearly always zero ... */
if (unlikely(error)) {
    /* ... */
}
```

Conversely, to mark a branch as very likely taken:

```
/* we predict 'success' is nearly always nonzero ... */
if (likely(success)) {
    /* ... */
}
```

You should only use these directives when the branch direction is overwhelmingly known *a priori* or when you want to optimize a specific case at the cost of the other case. This is an important point: These directives result in a performance boost when the branch is correctly marked, but a performance *loss* when the branch is mismarked. A common usage, as shown in these examples, for `unlikely()` and `likely()` is error conditions. As you might expect, `unlikely()` finds much more use in the kernel because `if` statements tend to indicate a special case.

No Memory Protection

When a user-space application attempts an illegal memory access, the kernel can trap the error, send the `SIGSEGV` signal, and kill the process. If the kernel attempts an illegal memory access, however, the results are less controlled. (After all, who is going to look after the kernel?) Memory violations in the kernel result in an *oops*, which is a major kernel error. It should go without saying that you must not illegally access memory, such as dereferencing a `NULL` pointer—but within the kernel, the stakes are much higher!

Additionally, kernel memory is not pageable. Therefore, every byte of memory you consume is one less byte of available physical memory. Keep that in mind the next time you need to add *one more feature* to the kernel!

No (Easy) Use of Floating Point

When a user-space process uses floating-point instructions, the kernel manages the transition from integer to floating point mode. What the kernel has to do when using floating-point instructions varies by architecture, but the kernel normally catches a trap and then initiates the transition from integer to floating point mode.

Unlike user-space, the kernel does not have the luxury of seamless support for floating point because it cannot easily trap itself. Using a floating point inside the kernel requires manually saving and restoring the floating point registers, among other possible chores. The short answer is: *Don't do it!* Except in the rare cases, no floating-point operations are in the kernel.

Small, Fixed-Size Stack

User-space can get away with statically allocating many variables on the stack, including huge structures and thousand-element arrays. This behavior is legal because user-space has a large stack that can dynamically grow. (Developers on older, less advanced operating systems—say, DOS—might recall a time when even user-space had a fixed-sized stack.)

The kernel stack is neither large nor dynamic; it is small and fixed in size. The exact size of the kernel's stack varies by architecture. On x86, the stack size is configurable at compile-time and can be either 4KB or 8KB. Historically, the kernel stack is two pages, which generally implies that it is 8KB on 32-bit architectures and 16KB on 64-bit architectures—this size is fixed and absolute. Each process receives its own stack.

The kernel stack is discussed in much greater detail in later chapters.

Synchronization and Concurrency

The kernel is susceptible to race conditions. Unlike a single-threaded user-space application, a number of properties of the kernel allow for concurrent access of shared resources and thus require synchronization to prevent races. Specifically

- Linux is a preemptive multitasking operating system. Processes are scheduled and rescheduled at the whim of the kernel's process scheduler. The kernel must synchronize between these tasks.
- Linux supports symmetrical multiprocessing (SMP). Therefore, without proper protection, kernel code executing simultaneously on two or more processors can concurrently access the same resource.
- Interrupts occur asynchronously with respect to the currently executing code. Therefore, without proper protection, an interrupt can occur in the midst of accessing a resource, and the interrupt handler can then access the same resource.
- The Linux kernel is preemptive. Therefore, without protection, kernel code can be preempted in favor of different code that then accesses the same resource.

Typical solutions to race conditions include spinlocks and semaphores. Later chapters provide a thorough discussion of synchronization and concurrency.

Importance of Portability

Although user-space applications do not *have* to aim for portability, Linux is a portable operating system and should remain one. This means that architecture-independent C code must correctly compile and run on a wide range of systems, and that architecture-dependent code must be properly segregated in system-specific directories in the kernel source tree.

A handful of rules—such as remain endian neutral, be 64-bit clean, do not assume the word or page size, and so on—go a long way. Portability is discussed in depth in a later chapter.

Conclusion

To be sure, the kernel has unique qualities. It enforces its own rules and the stakes, managing the entire system as the kernel does, are certainly higher. That said, the Linux kernel's complexity and barrier-to-entry is not qualitatively different from any other large soft-

ware project. The most important step on the road to Linux development is the realization that the kernel is not something to fear. Unfamiliar, sure. Insurmountable? Not at all.

This and the previous chapter lay the foundation for the topics we cover through this book's remaining chapters. In each subsequent chapter, we cover a specific kernel concept or subsystem. Along the way, it is imperative that you read and modify the kernel source. Only through actually reading and experimenting with the code can you ever understand it. The source is freely available—use it!

This page intentionally left blank

Index

64-bit atomic operations, 180-181

A

absolute time, 207

abstraction layer, VFS (Virtual Filesystem),
262-263

account_process_tick() function, 219

action modifiers, gfp_mask flags, 239-240

action string, Kernel Event Layer, 361

activate task() function, 61

address intervals

 creating, 318-320

 removing, 320

address_space object, page caches,
326-328

address_space operations, page caches,
328-330

*Advanced Programming in the UNIX
Environment*, 409

advisory locks, 166

AIX (IBM), 2

algorithms, 109-111

 asymptotic behavior, 109

 big-o notation, 109

 big-theta notation, 109-110

 clairvoyant, 325

 complexity, 109-110

 time complexity, 110-111

 listing of, 110-111

 process scheduler, 46-50

 scalability, 109

scheduling algorithms, priority-based scheduling, 44

alignment of data, 386-387

issues, 387
nonstandard types, 387
structure padding, 387-389

alloc_pages() function, 236, 259

alloc_page() function, 236

alloc_percpu() function, 258

allocating

memory, 237-244
memory descriptor, 308
process descriptors, 25-26
UIDs (unique identification numbers), 101-102
which method to use, 259

allocating memory, 231, 237, 260

choosing method, 259
high memory mappings, 253
permanent mappings, 254
temporary mappings, 254-255

kfree() function, 243-244

kmalloc() function, 238-244

gfp_mask flags, 238-243

pages, 231-232

obtaining, 235-237

per-CPU allocations, 255-256

slab layers, 245-246

design, 246-249

interface, 249-252

statically allocating on stack, 252-253

vmalloc() function, 244-245

zones, 233-235

allow interrupts flag, 127

anonymous mapping, 318

Anticipatory I/O scheduler, 302-303

APIC timer, 217

APIs

system calls, 70

UNIX Network Programming, 409

applications

hardware, relationship, 6

interrupt handlers, writing, 118-119

kernel, relationship, 6

arch directory, kernel source tree, 13

arguments, system calls, 71

arrays, per-CPU data, 255

***Art of Computer Programming, The, Volume 1*, 409**

assembly, inline assembly, 19

asserting bugs, 370-371

associative arrays. See maps

asymptotic behavior, algorithms, 109

asynchronous interrupts, 114

atomic context, 115

atomic high memory mappings, 254-255

atomic operations, synchronization methods, 175

64-bit operations, 180-181

bitwise operations, 181-183

converting, 177

counter implementation, 177

defining, 177

increments, 175-176

integer operations, 176-179

interfaces, 176

nonatomic bit operations, 183

overhead advantages, 179

testing, 177

atomic_t data type, 384

atomicity, ordering, compared, 179

B

Bach, Maurice, 407

backing stores, 323

balanced binary search trees, self-balanced
binary search trees

rbtrees, 106-108

red-black trees, 105-106

barrier operations, ordering, 179

barrier() function, 206

barriers

functions, 204-205

memory reads/writes, 203-206

bdflush kernel thread, 333-334

behaviors, system calls, 71-72

Bell Laboratories, Unix developmental
history, 1

Benvenuti, Christian, 408

Berkeley Software Distributions (BSD), 2

BH interface, tasklets, 148

bh_state flags (buffers), 292

big-endian byte ordering, 389-391

big-o notation, 109

big-theta notation, 109-110

binary searching, git source management
tool, 376-377

binary semaphores, 191-192

binary trees, 103-104

BSTs (binary search trees), 104

self-balanced binary search trees, 105

rbtrees, 106-108

red-black trees, 105-106

binding system calls, 79-81

bio structure, block I/O layer,
294-295

bitwise atomic operations, 181-183

BKL (Big Kernel Lock), 198-199

block device nodes, 337

block devices, 289-290, 337

buffer heads, 291

buffers, 291-294

sectors, 290-291

block directory, kernel source code, 13

block I/O layer, 290

bi_cnt field, 296

bi_idx field, 296

bi_io_vecs field, 295

bi_private field, 296

bi_vcvt field, 295

bio structure, 294-295

I/O vectors, 295-296

segments, 294

versus buffer heads, 296-297

blocks, 289-290, 337

BLOCK_SOFTIRQ tasklet, 140

BogoMIPS value, 227

Booleans, 14

Bostic, K., 408

bottom halves

disabling, 157-159

interrupt handlers, 115,
133-135

benefits, 134-135

BH interface, 135-136

task queues, 135

locking between, 157

mechanism selection criteria,
156-157

softirqs, 136-141

spin locks, 187-188

tasklets, 136, 142-148

version terminology, 137

work queues, 149-156

braces, coding style, 398-399

branch annotation, GNU C, 19-20

BSTs (binary search trees), 104

buffer caches, 330-331

buffers, blocks, 291-294

bug reports, submitting, 403-404

BUG() routine, 370

BUG_ON() routine, 370

bugs

asserting, 370-371

range of, 364

reproducing, 363-364

building

Booleans, 14-15

kernel, 13-16

modules, 340-342

noise minimization, 15

spawning multiple jobs, 16

busy looping, timers, 225-226

byte ordering, 389-391

C

C library, 5

system calls, 70-71

***C Programming Language, The*, 399, 409**

C++-style comments, 400

cache eviction, 324-325

cache hits, 323

cache misses, 323

caches, 246

cache miss, 323

caching

backing stores, 323

buffer caches, 330-331

cache eviction, 324-325

cache hits, 323

page cache, 324

page caches, 323-326

address_space object, 326-328

address_space operations, 328-330

global hash, 330

radix tree, 330

page caching, filesystem files, 326

write caching, 324

write-through caches, 324

cdevs. See character devices

CFQ (Complete Fair Queuing) I/O scheduler, 303

CFS Schedulers, 172

character device nodes, 337

character devices, 289, 337

characters, word size, 381

child tasks, reparenting, 38

Choffnes, David R., 407

circular linked lists, 86-87

clairvoyant algorithm, 325

classes, process scheduler, 46-47

cli() function, 128

clocks, real-time clock (RTC), 217

clone() function, flags, 34-35

clone() system call, 32-34

clusters, 290

coarse locking, 172

code, interrupt-safe code, 168

codes, locks, compared, 186

coding style

braces, 398-399

comments, 400-401

consistency, 396

existing routines, 402

fixing ex post facto, 403

functions, 400

ifdef preprocessor directives, 402

importance of, 396

- indentation, 396
- line length, 399-400
- naming conventions, 400
- productivity, 396
- spacing, 397-398
- structure initializers, 402-403
- switch statements, 396-397
- typedefs, 401
- commands**
 - modprobe, 343
 - SysRq, 371
- Comments, coding style, 400-401**
- community help resources, debugging, 377**
- complete() function, 198**
- Completely Fair Scheduler, 43**
- completion variables, 197-198**
- concurrency**
 - causes, 167
 - interrupts, 167
 - kernel, 21
 - kernel preemption, 167
 - pseudo-concurrency, 167
 - sleeping, 167
 - softirqs, 167
 - symmetrical multiprocessing, 167
 - tasklets, 167
 - true concurrency, 167
- concurrent programming, threads, 33**
- cond_resched() function, 226**
- condition variables, debugging, 374**
- conditionals, UIDs, 373-374**
- CONFIG options, 168**
- configuration, kernel, 14-15**
- configuration options, modules, managing, 344-346**
- congestion, avoiding with multiple threads, 334-335**

- contended threads, 184**
- contention, locks, 171**
- context**
 - interrupts, 115
 - processes, 29
 - system calls, 78-81
- context switch() function, 62**
- context_switch() method, 380**
- context switching, process scheduler, 62**
- controlling interrupts, 127-130**
- converting atomic operations, 177**
- Cooper, Chris, 408**
- cooperative multitasking, process scheduler, 41-42**
- copy-on-write (COW) pages, 31**
- copy_process() function, 32**
- Corbet, Jonathan, 408**
- counters, implementing, atomic operations, 177**
- counting semaphores, 191-192**
- COW (copy-on-write) pages, 31**
- CREDITS file, 403**
- critical regions, multiple threads of execution, 162**
- crypto directory, kernel source tree, 13**
- ctime() library call, 221**
- current date and time, 207, 220-221**
- CVS, 11**
- cylinders, 290**

D

- D-BUS, Kernel Event Layer, 361**
- data section (processes), 23**
- data structures**
 - binary trees, 103-104
 - BSTs (binary search trees), 104
 - self-balanced binary search trees, 105-108

- choosing, 108
- filesystems, 285–288
- freeing, slab layers, 245–252
- linked lists, 85
 - adding a node to, 90–91
 - circular linked lists, 86–87
 - defining, 89–90
 - deleting a node from, 91–92
 - doubly linked lists, 85–86
 - iterating through backward, 94
 - iterating while removing, 95
 - kernel implementation, 88–90
 - manipulating, 90–92
 - moving nodes, 92
 - navigating through, 87–88
 - singly linked lists, 85–86
 - splicing nodes, 92
 - traversing, 93–96
- maps, 100–101
 - UIDs (unique identification numbers), 100–103
- queues, 96–97
 - creating, 97–98
 - dequeuing data, 98
 - destroying, 99
 - enqueueing data, 98
 - kfifo, 97–100
 - obtaining size of, 98
 - resetting, 99
- VFS (Virtual Filesystem), 265–266

data types

- atomic_t, 384
- char, 386
- dev_t, 384
- explicitly sized data types, 385–386
- gid_t, 384
- opaque data types, 384

- pid_t, 384
- portability, 384
- special data types, 384–385
- uid_t, 384
- usage rules, 384

deactivating timers, 223**Deadline I/O scheduler, 300–302****deadlocks**

- ABBA, 170
- threads, 169–171

debuggers in-kernel debugger, 372–373**debugging, 363–364, 378**

- atomicity, 370
- binary searching, 376–377
- BUG() routine, 370
- bugs
 - asserting, 370–371
 - reproducing, 363–364
- community help resources, 377
- condition variables, 374
- difficulty of, 363
- dump information, 370–371
- dump stack() routine, 371
- kernel options, 370
- Magic SysRq key commands, 371–372
- occurrence limiting, 375–376
- oops, 367–369
 - kallsyms, 369–370
 - kysmoops, 369
- panic() routine, 371
- printing, 364–367
- rate limiting, 375–376
- spin locks, 186
- statistics, 374
- UID as a conditional, 373–374

declaring

- kobjects, 352-353
- linked lists, 88
- tasklets, 144-145

decoded version, oops, 369**deferences, 92****defining**

- atomic operations, 177
- linked lists, 89-90

Deitel, Harvey, 407-408**Deitel, Paul, 407****del_timer_sync() function, 223****delays, timers, 226-227****denoting system calls, 73-74****dentries, sysfs, 355****dentry object, VFS (Virtual Filesystem), 265, 275-276**

- caches, 276-277
- operations, 278-279
- states, 276

dequeuing data, 98**design, slab layers, 246-252*****Design and Implementation of the 4.4BSD Operating System, The*, 408*****Design of OS/2, The*, 408*****Design of the Unix Operating System, The*, 407****dev_t data type, 384****development kernel, 8-10**

- maintenance, 403

device model

- benefits, 348-349
- kobjects, 349-350
 - declaring, 352-353
 - embedding, 350
 - managing, 352-353
 - sysfs filesystem, 355-362

ksets, 351**ktypes, 350-351****name pointer, 349****parent pointer, 350****reference counts, 353-355**

- incrementing and decrementing, 354

kref structure, 354-355**sd pointer, 350****structures, 351-352****devices, 337****block devices, 289-290****buffer heads, 291****buffers, 291-294****sectors, 290-291****character devices, 289, 337****drivers, 114****glock devices, 337****miscellaneous devices, 338****network devices, 338****Dijkstra, Edsger Wybe, 192****directories, 264****directory object, VFS (Virtual Filesystem), 265****dirty lists, 324****dirty page writeback, 331****disable irq nosync() function, 129****disable irq() function, 129-130****disable_irq() function, 130****disable_irq_nosync() function, 130****disabling**

- bottom halves, 157-159
- interrupts, 127-129
- kernel preemption, 201-202

do mmap() function, 318-319**do softirq() function, 138-141****do timer() function, 218**

documentation

- coding style, 396
- self-generating documentation, 401

Documentation directory, kernel source tree, 13

doublewords, 382

doubly linked lists, 85-86

down interruptible() function, 193-194

down trylock() function, 193-194

down() function, 194

downgrade write() function, 195

do_exit() function, 36

do_IRQ() function, 123-125

do_munmap() function, 320

do_timer() function, 218

drivers, 114

- RTC (real-time clock) driver, 120-122

drivers directory, kernel source tree, 13

dump information, debugging, 370-371

dump_stack() function, 371

dynamic timers, 207, 222

E

early printk() function, 365

elements, 85

elevators, I/O schedulers, 299-300

embedding kobjects, 350

enable_irq() function, 130

enabling interrupts, 127-128

enqueueing data, 98

entity structure, process scheduler, 50

entry points, scheduler, 57-58

epoch, 220

Ethernet devices. See network devices

events, relationship with time, 207

eviction (cache), 324-325

exceptions, 114

exec() function, 31

executable files, 29

execution, softirqs, 138-140

exokernel, 7

Expert C Programming, 409

explicitly sized data types, 385-386

exported symbols, modules, 348

F

fair scheduling, 48-50

family tree, processes, 29-30

fields, memory descriptor, 307-308

file attributes, kobjects, 358-359

- conventions, 360-361

- creating, 359-360

- destroying, 360

file metadata, 264

file object, VFS (Virtual Filesystem), 265, 279-280

- operations, 280-284

file-backed mapping, 318

files, 263

- header files, 17

- kobjects, adding to, 358-361

- metadata, 264

filesystem

- abstraction layer, 262-263

- interface, 261-262

- UNIX filesystems, 264

filesystem blocks, 290

filesystem files, page caching, 326

filesystem interface, 261

filesystems, 263, 264. See also VFS (Virtual Filesystem)

- data structures, 285-288

- Linux, support, 288

- metadata, 264

- UNIX filesystems, 263
- VFS (Virtual Filesystem)
 - data structures, 265-266
 - objects, 265-266
- files_struct data structure, 287**
- find_get_page() method, 329**
- find_vma() function, 316-317**
- find_vma_prev() function, 317**
- find_vma_intersection() function, 317**
- firmware directory, kernel source code, 13**
- fixed-size stacks, 20**
- flags**
 - clone() function, 34-35
 - interrupt handlers, 116-117
 - map type flags, 319
 - page protection flags, 319
 - VMAs (virtual memory areas), 311-312
- flat address spaces, 305**
- floating point instructions, 20**
- flush scheduled work() function, 154**
- flusher threads, 331-335**
- flushing work queues, 154**
- fork() function, 24, 31-34**
- forking, 32**
- free lists, 245**
- free_percpu() function, 258**
- free_irq() function, 118**
- freeing**
 - data structures, slab layers, 245-252
 - interrupt handlers, 118
- freeing pages, 237**
- frequencies, timer interrupts, 209**
- front/back merging, I/O scheduler, 299-300**
- fs directory, kernel source tree, 13**

- fs_struct data structure, 287**

- ftime() library call, 221**

functions

- account_process_tick(), 219
- cli(), 128
- clone(), 34-35
- coding style, 400
- context_switch(), 62
- copy_process(), 32
- disable_irq(), 129-130
- disable_irq_nosync(), 130
- do_exit(), 36
- do_IRQ(), 123-125
- do_mmap(), 318-320
- do_munmap(), 320
- do_softirq(), 138
- enable_irq(), 130
- exec(), 31
- find_vma_prev(), 317
- find_vma(), 316-317
- find_vma_intersection(), 317
- fork(), 31-32, 34
- free_irq(), 118
- hello_init(), 339
- idr_destroy(), 103
- inline functions, 18-19, 400
- in_interrupt(), 130
- in_irq(), 130
- irqs_disabled(), 130
- kfree() function, 243-244
- kmalloc(), 238-244
 - gfp_mask flags, 238-243
- kthread_create(), 36
- likely(), 20
- list_add(), 91
- list_del(), 91
- list_for_each(), 93

list_for_each_entry(), 96
 list_move(), 92
 list_splice(), 92
 local_bh_disable(), 157
 local_irq_disable(), 130
 local_irq_enable(), 130
 local_irq_restore(), 130
 local_irq_save(), 130
 malloc(), 238
 mmap(), 319-320
 munmap(), 320
 nice(), 66
 open(), 5
 panic(), 371
 printf(), 5, 17, 364-367
 printk(), 17, 364-367, 375
 raise_softirq(), 141
 read(), 326
 relationship with time, 207
 request_irq(), 118
 schedule_timeout(),
 227-230
 strcpy(), 5
 tasklet_disable(), 145
 tasklet_disable_nosync(), 145
 tasklet_enable(), 146
 tasklet_kill(), 146
 tick_periodic(), 219
 unlikely(), 20
 update_curr(), 51-52
 vfork(), 33-34
 vmalloc(), 244-245
 void local_bh_disable(), 158
 void local_bh_enable(), 158
 wait(), 24
 wake_up_process(), 36
 write(), 5

G

Gagne, Greg, 407
Galvin, Peter Baer, 407
gcc (GNU Compiler Collection), 18
gdb, 373
generating patches, 404-405
get bh() function, 293
get cpu() function, 202
get sb() function, 285
get_cpu_var() function, 258
get_free_page() function, 236
get_zeroed_page() function, 237
gettimeofday() function, 221
gettimeofday() system call, 221
gfp_mask flags, kmalloc() function, 238-243
gid_t data type, 384
git source management tool, 11-12
 binary searching, 376-377
 generating patches, 405
global hash, page caches, 330
global variables, jiffies, 212-216
GNU C, 18
 branch annotation, 19-20
 inline assembly, 19
 inline functions, 18-19
GNU debugger, 372-373
GNU General Public License (GPL), 4
Göüdel, Escher, Bach, 409
granularity, locking, 171

H

hackers, 403
HAL (hardware abstraction layer), 357
halves
 division of work, 134
 interrupt handlers, 115-116

handlers, system calls, 73-74

hard real-time scheduling policies, 64

hard sectors. *See* sectors

hardware, applications, relationship, 6

header files, 17

heads, 290

Hello, World! module, 338-340

hello_init() function, 339

HI_SOFTIRQ tasklet, 140

high memory, 393

high memory mappings, 253-255

hitting, timers, 208

Hofstadter, Douglas, 409

HP-UX (Hewlett Packard), 2

HP-UX 11i Internals, 408

HRTIMER_SOFTIRQ tasklet, 140

Hungarian notation, 400

Hz values, 208-212

- jiffies global variable, 216

I/O block layer, request queues, 297

I/O blocks, 290

I/O schedulers, 297-298

- Anticipatory I/O scheduler, 302-303
- CFQ (Complete Fair Queuing) I/O scheduler, 303
- Deadline I/O scheduler, 300-302
- front/back merging, 299-300
- Linux Elevator, 299-300
- merging/sorting functions, 298-299
- minimized read latency, 302-303
- Noop I/O scheduler, 303-304
- request starvation prevention, 300-302
- selection options, 304

I/O-bound processes, versus processor-bound processes, 43-44

idle process, operating systems, 6

idr_destroy() function, 103

IEEE (Institute of Electrical and Electronics Engineers), 70

ifdef preprocessor directives, coding style, 402

implementation

- interrupt handlers, 123-126
- softirqs, 137-140
- system calls, 74-78
- tasklets, 142-144
- timers, 224
- work queues, 149-153

implementing system calls, 82-83

in interrupt() function, 130

in-kernel debugger, 372-373

in_interrupt() function, 130

in_irq() function, 130

include directory, kernel source tree, 13

incremental patches, 12

increments, atomic operations, 175-176

indent utility, 403

indentation, coding style, 396

indexes, softirqs, 140-141

init_completion() function, 198

init directory, kernel source tree, 13

initialization, semaphores, 192

inline functions, 400

- GNU C, 18-19

inode, 264

inode object, VFS (Virtual Filesystem), 265, 270-274

inodes, page caches, 331

installation

- kernel, 16
- modules, 342
- source code, 12

integer atomic operations, 176-179

- 64-bit atomic operations, 180-181

interfaces

- atomic operations, 176
- filesystem, 261-262
- slab layers, 249-252
- wrapping, 402

internal representation, jiffies global variable, 213-214**internal values, timers, 222****interprocess communication (IPC) mechanism, 7****interrupt context, 5**

- kernels, 122
- stack space, 122-123

interrupt handlers, 5, 113

- bottom halves, 115-116, 133-135
 - benefits, 134-135
 - BH interface, 135-136
 - softirqs, 136-141
 - task queues, 135
 - tasklets, 136
- controlling interrupts, 127-130
- do_IRQ() function, 123-125
- flags, 116-117
- freeing, 118
- free_irq() function, 118
- function of, 114-115
- implementing, 123-126
- interrupt-safe code, 168
- limitations, 133
- locks, 185-186
- reentrancy, 119
- registering, 116
- request_irq() function, 118
- RTC (real-time clock) driver, 120-122
- shared, 119-120

- speed of, 122
- timer, 217-220
- top half, 115
- top halves, 133
- when to use, 135
- writing, 118-119

interrupt request (IRQ), 114**interrupt service routine (ISR). See interrupt handlers****interrupt stacks, 122****interrupt-safe code, 168****interrupts, 5, 113-114, 117, 131**

- asynchronous, 114
- concurrency, 167
- context, 115
- controlling, 127-130
- disable irq nosync() function, 130
- disabling, 127-129
- enable irq() function, 130
- enabling, 127-128
- in interrupt() function, 130
- in irq() function, 130
- irqs disabled() function, 130
- local irq disable() function, 130
- local irq enable() function, 130
- local irq save() function, 130
- synchronous, 114
- timers, frequencies, 209

ioctl() method, 284**IPC (interprocess communication) mechanism, 7****ipc directory, kernel source tree, 13****IRIX (SGI), 2****IRQ (interrupt request), 114****irqs_disabled() function, 130****ISR (interrupt service routine), 114****iterating linked lists, 94-95**

J

jiffies, 391

- origins of term, 212-213
- sequential locks, 200

jiffies global variable, 212-213

- HZ values, 216
- internal representation, 213-214
- wraparounds, 214-216

K

kallsyms, 369-370**Karels, Michael J., 408****kbuild build system, building modules, 340-342****KERN ALERT loglevel, printk() function, 366****KERN CRIT loglevel, printk() function, 366****KERN DEBUG loglevel, printk() function, 366****KERN EMERG loglevel, printk() function, 366****KERN ERR loglevel, printk() function, 366****KERN INFO loglevel, printk() function, 366****KERN NOTICE loglevel, printk() function, 366****KERN WARNING loglevel, printk() function, 366****kernel**

- applications, relationship, 6
- building, 13-16
- C library, 17
- concurrency, 21
- configuring, 14-15
- debugging help resources, 377
- defined, 4
- development kernel, 8-10
- downloading, 11
- fixed-size stack, 20
- floating point instructions, 20
- hardware, 5
 - relationship, 6

- implementing, linked lists, 88-90

- installing, 16

- interrupt context, 5

- interrupt handlers, 5

- lack of memory protection, 20

- modules, 7

- monolithic, 7

- naming conventions, 9

- portability, 21

- preemption, concurrency, 167

- producer and consumer pattern, 96

- root directories, 12-13

- rules, 16-21

- small, fixed-size, 21

- source tree, 12-13

- stable kernel, 8-9, 11

- structure, 88

- synchronization, 21

- system calls, 71

- vendor kernels, 14

kernel directory, kernel source tree, 13**Kernel Event Layer**

- D-BUS, 361

- kobjects, 361-362

- netlink, 361

- parameters, 362

- payloads, 361

- verb strings, 361

kernel locked() function, 199**kernel maintainer, 403****kernel messages**

- klogd daemon, 367

- log buffer, 366-367

- oops, 367-370

- syslogd daemon, 367

Kernel Newbies website, 395

kernel objects, 337

kernel preemption, 7, 393

per-CPU data, 256

process scheduler, 63-64

kernel random number

generator, 338

kernel threads, 35-36

memory descriptor, 309

pdflush task, 35

kernel timers. See timers

Kernel Traffic website, 395

kernel-space, 29

Kernel.org, 409

Kernighan, Brian, 399, 409

kfifo queues, 97-100

creating, 97-98

dequeuing data, 98

destroying, 99

enqueueing data, 98

obtaining size of, 98

resetting, 99

kfree() function, 243-244

kgdb, 373

klogd daemon, kernel messages, 367

kmalloc() function, 238-244, 259

gfp_mask flags, 238-243

Knuth, Donald, 409

kobjects

device model, 349-350

managing, 352-353

file attributes, 358-359

conventions, 360-361

creating, 359-360

destroying, 360

sysfs filesystem, 355

adding and removing from, 357-358

adding files, 358-361

dentries, 355

Kernel Event Layer, 361-362

root directories, 357

kobject_create() function, 353

Kogan, Michael, 408

kqdb debugger, 373

kref structure, device model reference counts, 354-355

kref_put() function, 354

Kroah-Hartman, Greg, 408

ksets, device model, 351

ksoftirqd task, 35

ksoftirqd threads, tasklets, 146-147

kthreadd kernel process, 36

kthread_create() function, 36

ktypes, device model, 350-351

kupdated kernel thread, 333-334

kysmoops, 369

L

laptop mode, page writeback, 333

last-in/first-out (LIFO) ordering, 94

least recently used (LRU), cache eviction, 325

lib directory, kernel source tree, 13

libc functions, 17

lifecycle, processes, 24

lightweight processes, threads, 34

likely() function, 20

limitations, interrupt handlers, 133

line length, coding style, 399-400

linked lists, 85

circular linked lists, 86-87

- declaring, 88
 - defining, 89-90
 - doubly linked lists, 85-86
 - iterating through backward, 94
 - iterating while removing, 95
 - kernel implementation, 88-90
 - manipulating, 90-92
 - memory, 313
 - navigating through, 87-88
 - nodes
 - adding to, 90-91
 - deleting from, 91-92
 - moving, 92
 - splicing, 92
 - singly linked lists, 85-86
 - traversing, 93-96
- Linux Elevator, I/O schedulers, 299-300**
- Linux, 1**
- development history, 3
 - dynamic loading, 8
 - filesystems, support, 288
 - kernel development community, 10
 - object-oriented device model, 8
 - open source status, 4
 - portability, 380-381
 - preemptive nature, 8
 - scalability, 171
 - symmetrical multiprocessor (SMP), 8
 - thread implementation, 33-36
 - thread support, 8
 - Unix, 3
 - versus Unix kernel, 6, 8
- Linux Device Drivers, 408**
- Linux kernel community, 395**
- Linux Kernel Mailing List (lkml), 10, 395**
- Linux System Programming, 409**
- Linux Weekly News, 395, 409**
- list for each() function, 93**
 - list move() function, 92**
 - list splice() function, 92**
 - lists, VMAs (virtual memory areas), 313-314**
 - list_add() function, 91**
 - list_del() function, 91**
 - list_for_each_entry() function, 96**
 - little-endian byte ordering, 389-391**
 - lkml (Linux Kernel Mailing List), 10, 395**
 - loading**
 - modules, 343-344
 - managing configuration options, 344-346
 - local bh disable() function, 157**
 - local bh enable() function, 157-158**
 - local_irq_disable() function, 130**
 - local_irq_enable() function, 130**
 - local_irq_restore() function, 130**
 - local_irq_save() function, 130**
 - lock contention, 171**
 - lock kernel() function, 199**
 - locking**
 - coarse locking, 172
 - granularity, 171
 - need of protection, 168-169
 - race conditions, 165-166
 - locking between bottom halves, 157**
 - locks, 165**
 - acquiring, 193
 - advisory, 166
 - BKL (Big Kernel Lock), 198-199
 - busy wait, 166
 - contention, 171
 - deadlocks, threads, 169-171
 - debugging, 186
 - functions, 193
 - mutexes, 195-197

- non-recursive nature, 185
- releasing, 193
- semaphores, 190-191
 - binary semaphores, 191-192
 - counting semaphores, 191-192
 - creating, 192-193
 - implementing, 193-194
 - initializing, 192
 - reader-writer semaphores, 194-195
- sequential locks, 200-201
- spin locks, 183-187
 - bottom halves, 187-188
 - debugging, 186
 - methods, 184-187
 - reader-writer spin locks, 188-190
- use in interrupt handlers, 185-186
- versus code, 186
- voluntary, 166
- log buffers, kernel messages, 366-367**
- loglevels, printk() function, 365-366**
- looking up UIDs (unique identification numbers), 102-103**
- Love, Robert, 409**
- LRU (least recently used), cache eviction, 325**

M

- Mac OS X Internals: A Systems Approach, 408***
- Magic SysRq key commands, 371-372**
- maintainers, 403**
- malloc() function, 238, 306**
- map type flags, 319**
- mapping, 100**
 - anonymous mapping, 318
 - file-backed mapping, 318
 - VMA's (virtual memory areas), 312
- mappings (high memory), 253**
 - permanent mappings, 254
 - temporary mappings, 254-255
- maps, UIDs (unique identification numbers), 100**
 - allocating, 101-102
 - looking up, 102
 - removing, 103
- Mauro, Jim, 408**
- mb() function, 204-205**
- McCreight, Edward M., 327**
- McDougall, Richard, 408**
- McKusick, Marshall Kirk, 408**
- mdelay() function, 227**
- memory**
 - allocation, 231, 260
 - choosing method, 259
 - high memory mappings, 253-255
 - kfree() function, 243-244
 - kmalloc() function, 238-244
 - pages, 231-232, 235-237
 - per-CPU allocations, 255-258
 - slab layers, 245-252
 - statically allocating on stack, 252-253
 - vmalloc() function, 244-245
 - zones, 233-235
 - high memory, 393
 - linked list, 313
 - memory areas, 305-306
 - memory descriptor, 306
 - mmap field, 313
 - MMUs (memory management units), 231
 - objects, pinned, 353

- pages, 231-233
 - freeing, 237
 - obtaining, 235-244
 - zeroed pages, 236-237
 - zones, 233-235
- process address space, 305
- red-black tree, 313
- VMAs (virtual memory areas), 309-310, 314-315
 - flags, 311-312
 - lists, 313-314
 - locating, 316-317
 - operations, 312-313
 - private mapping, 312
 - shared mapping, 312
 - trees, 313-314
- memory areas, 314-315. See also VMAs (virtual memory areas)**
 - lists, 313-314
 - manipulating, 315-318
 - trees, 313-314
- memory descriptor, 306**
 - allocating, 308
 - destroying, 309
 - fields, 307-308
 - kernel threads, 309
 - mm struct, 309
- memory maps, 306**
- memory-management unit (MMU), 6**
- memory protection, kernel, lack of, 20**
- memory reads/writes, 203-206**
- memset() function, 353**
- merging functions, I/O scheduler, 298-299**
- message passing, 7**
- metadata files, 264**
- methods**
 - context_switch(), 380
 - ioctl(), 284
 - readpage(), 328
 - spin locks, 184-187
 - switch_mm(), 380
 - switch_to(), 380
 - synchronization methods, 175
 - 64-bit atomic operations, 180-181
 - atomic operations, 175-179
 - barriers, 203-206
 - bitwise atomic operations, 181-183
 - BKL (Big Kernel Lock), 198-199
 - completion variables, 197-198
 - mutexes, 195-197
 - nonatomic bit operations, 183
 - ordering, 203-206
 - preemption disabling, 201-202
 - semaphores, 190-195
 - sequential locks, 200-201
 - spin locks, 183-190
 - writepage(), 328
- microkernel designs, monolithic designs, compared, 7**
- microkernels, message passing, 7**
- migration threads, 66**
- miscellaneous devices, 338**
- mm directory, kernel source tree, 13**
- mm struct, memory descriptor, 309**
- mmap() function, 306, 319**
- MMUs (memory management units), 6, 231**
- mod timer() function, 223**
- Modern Operating Systems*, 407**
- modprobe command, 343**
- modules, 14, 337-338**
 - building, 340-342
 - configuration options, managing, 344-346
 - dependencies, generating, 342

- exported symbols, 348
- Hello, World!, 338-340
- installing, 342
- kernel, 7
- living externally of kernel source tree, 342
- loading, 343-344
- parameters, 346-347
- removing, 343
- source trees, 340-342
- MODULE_AUTHOR() macro, 340**
- MODULE_DESCRIPTION() macro, 340**
- module_exit() function, 339**
- module_init() macro, 339**
- MODULE_LICENSE() macro, 340**
- monolithic kernel, microkernel designs, compared, 7
- Moore, Chris, 408
- Morton, Andrew, 9
- mount flags, 286
- mount points, 263
- multiplexing system calls, 74
- multiprocessing, symmetrical multiprocessing, 161
 - concurrency, 167
- multitasking, 41-42
- munmap() function, 320**
- mutexes, 191, 195-197**

N

- name pointer, device model, 349**
- namespace data structure, 287-288**
- namespaces, 263**
- naming conventions**
 - coding style, 400
 - kernel, 9
- net directory, kernel source tree, 13**

- NET_RX_SOFTIRQ tasklet, 140**
- NET_TX_SOFTIRQ tasklet, 140**
- netlink, Kernel Event Layer, 361**
- network devices, 338**
- Neville-Neil, George V., 408**
- nice values, processes, 44**
- nice() function, 66**
- nodes, 85**
 - linked lists
 - adding to, 90-91
 - deleting from, 91-92
 - moving, 92
 - splicing, 92
- nonatomic bit operations, 183**
- Noop I/O scheduler, 303-304**
- notation, Hungarian notation, 400**
- numbers, system calls, 72**

O

- O(1) scheduler, 42-43**
- object-oriented device model, Linux, 8**
- objects**
 - pinned, 353
 - VFS (Virtual Filesystem), 265-266
 - dentry, 265, 275-279
 - directory, 265
 - file, 265, 279-284
 - inode, 265, 270-274
 - operations, 265
 - superblock, 265-269
- occurrence limiting, debugging, 375-376**
- oops, kernel messages, 367-370**
- opaque data types, 384**
- operations, VMAs (virtual memory areas), 312-313**
- open softirq() function, 141**
- open() function, 5**

open() system call, 261

Operating System Concepts, 407

operating systems, 4

- general activities, 5
- idle process, 6
- kernel-space, 5
- multitasking, 41
- portability, 379-380
- scalability, 171
- supervisor, 4
- system calls, 5
- tickless operations, 212

Operating Systems, 407

Operating Systems: Design and Implementation, 407

operations object, VFS (Virtual Filesystem), 265

order preservation, 100

ordering

- atomicity, compared, 179
- barrier operations, 179
- memory reads/writes, 203-206

OS News. com, 409

P

PAE (Physical Address Extension), 253

page caches, 323-326

- address_space object, 326-328
- address_space operations, 328-330
- buffer caches, 330-331
- filesystem files, 326
- flusher threads, 331-335
- global hash, 330
- radix tree, 330
- readpage() method, 328
- writepage() method, 328

page_count() function, 232

page global directory (PGD), 321

page middle directory (PMD), 321

page protection flags, 319

page size, architectures, 391-392

page tables, 320-322

- future management possibilities, 322
- levels, 320-321

page writeback, 323

- bdflush kernel thread, 333-334
- dirty page writeback, 331
- kupdated kernel thread, 333-334
- laptop mode, 333
- pdflush kernel thread, 333-334
- settings, 332

pageable kernel memory, 8

pages (memory), 231-233

- freeing, 237
- obtaining, 235-236
 - kfree() function, 243-244
 - kmalloc() function, 238-244
 - vmalloc() function, 244-245
 - zeroed pages, 236-237
- word size, 381
- zones, 233-235

panic() function, 371

parallelism, threads, 33

parameter passing, system calls, 74

parameters

- Kernel Event Layer, 362
- modules, 346-347
- system calls, verifying, 75-78

parent pointer, device model, 350

parentless tasks, 38-40

patches

- generating, 404-405
- incremental, 12
- submitting, 406

- payloads, Kernel Event Layer, 361**
- pdflush kernel thread, 333-334**
- pdflush task, 35**
- per-CPU allocations, 255-256**
 - percpu interface, 256-258
- per-CPU data**
 - benefits, 258-259
 - thrashing the cache, 258
- percpu interface, 256-258**
 - at compile-time, 256-257
 - at runtime, 257-258
- performance, system calls, 72**
- permanent high memory mappings, 254**
- PGD (page global directory), 321**
- PID (process identification), 26**
- pid_t data type, 384**
- pinned objects, 353**
- PIT (programmable interrupt timer), 217**
- PMD (page middle directory), 321**
- Pointers, dereferences, 92**
- policy (scheduler), 43-46**
 - I/O-bound processes, 43-44
 - priority-based scheduling, 44
 - processor-bound processes, 43-44
 - timeslices, 45
- poll() system call, 211**
- polling, 113**
- popping, timers, 208**
- portability, 21, 379**
 - byte ordering, 389-391
 - data alignment, 386-389
 - data types, 384
 - high memory, 393
 - implications of, 393
 - kernel preemption, 393
 - Linux, 380-381
 - operating systems, 379-380
 - page size architecture, 391
 - processor ordering, 392
 - scheduler, 380
 - SMP (symmetrical multiprocessing), 393
 - time, 391
 - word size, 381-384
- POSIX, system calls, 70**
- preempt count() function, 202**
- preempt disable() function, 202**
- preempt enable no resched() function, 202**
- preempt enable() function, 202**
- preemption**
 - kernel, concurrency, 167
 - process scheduler, 62
 - kernel preemption, 63-64
 - user preemption, 62-63
- preemption disabling, 201-202**
- preemptive multitasking, process scheduler, 41**
- printf() function, 5, 17, 364**
 - loglevels, 365-366
 - transposing, 367
- printing, debugging, 364-367**
- printk() function, 17, 375**
 - debugging, 364-366
 - loglevels, 365-366
 - nonrobustness of, 365
 - robustness of, 365
 - transposing, 367
- priority-based scheduling, 44**
- private mapping, VMAs (virtual memory areas), 312**
- /proc/interrupts file, 126-127**
- process address space**
 - address intervals
 - creating, 318-319
 - removing, 320

- flat versus segmented, 305
- memory areas, manipulating, 315-318
- memory descriptors, 306-308
 - allocating, 308
 - destroying, 309
 - kernel threads, 309
 - mm struct, 309
- overview, 305
- page tables, 320-322
- VMAs (virtual memory areas), 309-310, 314-315
 - flags, 311-312
 - lists, 313-314
 - operations, 312-313
 - trees, 313-314
- process descriptors**
 - allocating, 25-26
 - states, 27-29
 - storing, 26-27
 - task list, 24
 - TASK_INTERRUPTIBLE
 - process, 27
 - TASK_RUNNING process, 27
 - TASK_STOPPED process, 28
 - TASK_UNINTERRUPTIBLE
 - process, 28
- process descriptors (task list), 24-25**
- process scheduler, 41**
 - algorithm, 46-50
 - classes, 46-47
 - Completely Fair Scheduler
 - scheduler, 43
 - context switching, 62
 - cooperative multitasking, 41-42
 - entity structure, 50
 - entry point, 57-58
 - evolution, 42-43
 - fair scheduling, 48-50
 - implementing, 50-59, 61
 - O(1) scheduler, 42-43
 - policy, 43-46
 - I/O-bound processes, 43-44
 - priority-based scheduling, 44
 - processor-bound processes, 43-44
 - timeslices, 45
 - preemption, 62-64
 - preemptive multitasking, 41
 - process selection, 52-57
 - real-time scheduling policies, 64-65
 - Rotating Staircase Deadline
 - scheduler, 43
 - system calls, 65-67
 - time accounting, 50-52
 - timeslices, 42
 - Unix systems, 47-48
 - virtual runtime, 51-52
 - yielding, 42
- process states, 27-29**
- processes**
 - adding to trees, 54-55
 - address space, 23
 - context, 29
 - creating, 31
 - data structures, 286-288
 - defined, 23
 - I/O-bound processes, 43-44
 - lifecycle of, 24
 - nice values, 44
 - real-time, 44
 - real-time processes, 44
 - removing from trees, 56-57
 - resources, 23-24
 - runnable processes, 41
 - scalability, 171
 - task list, 24

- tasks, 24
- terminating, 24, 36-40
- threads, 305
- timeslice count, 211
- virtual memory, 23
- virtual processor, 23

- processor affinity system calls, 66
- processor ordering, 392
- processor time, yielding, 66
- processor-bound processors versus I/O-bound processes, 43-44
- profs virtual filesystem, 126-127
- producer and consumer programming pattern, kernel, 96
- programs, processes, 24
- pseudo-concurrency processes, 167
- put_bh() function, 293
- put_cpu_var() function, 258

Q

- quantum slice. *See* timeslices
- Quarterman, John S., 408
- queues, 96-97
 - creating, 97-98
 - dequeuing data, 98
 - destroying, 99
 - enqueueing data, 98
 - kfifo, 97-100
 - obtaining size of, 98
 - resetting, 99

R

- race conditions
 - ATM processing example, 163
 - locking, 165-166
 - multiple threads of execution, 162
 - timers, 224

- radix trees, page caches, 330
- Rago, Stephen, 409
- raise softirq irqoff() function, 141
- raise softirq() function, 141
- rate limiting, debugging, 375-376
- rbtrees, 106-108
- RCU_SOFTIRQ tasklet, 140
- read_barrier_depends() function, 204-205
- read lock irq() function, 189
- read lock irqsave() function, 189
- read lock() function, 189
- read_seqbegin() function, 220
- read_seqretry() function, 220
- read_unlock_irq() function, 189
- read_unlock_irqrestore() function, 189
- read_unlock() function, 189
- read() function, 326
- read() system call, 261
- reader-writer semaphores, 194-195
- reader-writer spin locks, 188-190
- readpage() method, 328
- read_barrier_depends() function, 205
- real-time clock (RTC) driver, 120-122, 217
- real-time priority, 44
- real-time scheduling policies, 64-65
- red-black binary trees, 105-106
- red-black trees, memory, 313
- reentrancy, interrupt handlers, 119
- reference counts, device model, 353-355
- registration, interrupt handlers, 116
- relative time, 207
- reparenting child tasks, 38
- REPORTING-BUGS file, 404
- request queues, I/O block layer, 297
- request_irq() function, 118
- Ritchie, Dennis, 1-3, 399, 409
- rmb() function, 204-205

root directories, sysfs file system, 357
 Rotating Staircase Deadline scheduler, 43
 routines, coding style, 402
 RTC (real-time clock) driver, 120-122, 217
 Rubini, Alessandro, 408
 rules, kernel, 16-21
 run_local_timers() function, 219
 run_local_timers() function, 224
 run_timer_softirq() function, 224
 runnable processes, 41
 Russinovich, Mark, 408
 rw lock init() function, 190

S

samples directory, kernel source code, 13
 scalability, 171

- algorithms, 109

 sched_getaffinity() system call, 66
 sched_getparam() system call, 66
 sched_getscheduler() system call, 66
 sched_get_priority_max() system call, 66
 sched_get_priority_min() system call, 66
 sched_setaffinity() system call, 66
 sched_setparam() system call, 66
 sched_setscheduler() system call, 66
 SCHED_SOFTIRQ tasklet, 140
 sched_yield() system call, 66-67
 schedule delayed work() function, 154-155
 scheduler, 41

- algorithm, 46-50
- classes, 46-47
- Completely Fair Scheduler scheduler, 43
- context switching, 62
- cooperative multitasking, 41-42
- entity structure, 50
- entry point, 57-58
- evolution, 42-43
- fair scheduling, 48-50
- implementing, 50-61
- O(1) scheduler, 42-43
- policy, 43-46
 - I/O-bound processes, 43-44
 - priority-based scheduling, 44
 - processor-bound processes, 43-44
 - timeslices, 45
- preemption, 62
 - kernel preemption, 63-64
 - user preemption, 62-63
- preemptive multitasking, 41
- process selection, 52-57
- real-time scheduling policies, 64-65
- Rotating Staircase Deadline scheduler, 43
- system calls, 65-67
- time accounting, 50-52
- timeslices, 42
- Unix systems, 47-48
- virtual runtime, 51-52
- yielding, 42

 schedule_timeout() function, 227-230
 scheduler_tick() function, 218-219
 scheduling

- tasklets, 143-146
- work queues, 153-154

 Schimmel, Curt, 408
 scripts directory, kernel source tree, 13
 sd pointer, device model, 350
 sectors, block devices, 290-291
 security directory, kernel source tree, 13
 segmented address spaces, 305
 segments, block I/O layer, 294-295
 select() system call, 211

self-balanced binary search trees, 105

- rbtrees, 106-108
- red-black trees, 105-106

self-generating documentation, 401**sema init() function, 193****semaphores, 190-191**

- binary semaphores, 191-192
- counting semaphores, 191-192
- creating, 192-193
- implementing, 193-194
- initializing, 192
- mutexes, compared, 197
- reader-writer semaphores, 194-195
- upping, 192

seqlocks, 220**Sequent DYNIX/ptx, 2****sequential locks, 200-201****settimeofday() system call, 221****settings, page writeback, 332****shared interrupt handlers, 119-120****shared mapping, VMAs (virtual memory areas), 312*****SIAM Journal of Computing*, 327****side effects, system calls, 71****Silberschatz, Abraham, 407****Singh, Amit, 408****single-page kernel stacks, statically allocating memory, 252-253****singly linked lists, 85-86****slab allocator, 25****"Slab Allocator: An Object-Caching Kernel Memory Allocator," 246****slab layers**

- design of, 246
- inode data structure example, 247-249
- interface, 249-252
- memory allocation, 245-252
- tenets of, 246

sleep, wait queues, 229**sleeping concurrency, 167****sleeping locks, 192**

- behaviors, 191
- mutexes, 195-197
 - versus semaphores, 197
 - versus spin locks, 197
- semaphores, 190-191
 - binary semaphores, 191-192
 - counting semaphores, 191-192
 - creating, 192-193
 - implementing, 193-194
 - initializing, 192
 - reader-writer semaphores, 194-195
 - versus spin locks, 191

SMP (symmetrical multiprocessing), 8

- portability, 393

smp mb() function, 205-206**smp read barrier depends() function, 205****smp rmb() function, 205-206****smp wmb() function, 205-206****smp_read_barrier_depends() function, 206****soft real-time scheduling policies, 64****softirqs**

- assigning indexes, 140-141
- bottom half mechanism, 137-138
- bottom half mechanism, executing, 140
- bottom half mechanism, index assignments, 140
- bottom halves, 136-141, 188
- concurrency, 167
- executing, 138-140
- handler, 138
- handlers, registering, 141
- implementing, 137-140
- ksoftirqd threads, 146-147
- raising, 141
- types, 140

Solaris (Sun), 2

Solaris Internals: Solaris and OpenSolaris Kernel Architecture, 408

Solomon, David, 408

sorting functions, I/O scheduler, 298-299

sound directory, kernel source tree, 13

source code, 11-12

source trees, 12-13

modules, 340-342

spacing coding style, 397-398

special data types, 384-385

spin is locked() method, 187

spin lock init() method, 186

spin lock irq() function, 186

spin lock irqsave() method, 187

spin locks, 183-186

bottom halves, 187-188

debugging, 186

methods, 184-187

mutexes, compared, 197

reader-writer spin locks, 188-190

spin try lock() method, 186

spin unlock() method, 187

spin_is_locked() method, 187

spin_lock() method, 187

spin_lock_init() method, 187

spin_lock_irq() method, 186

spin_lock_irqsave() method, 185

spin_trylock() method, 187

spin_unlock_irq() method, 187

spin_unlock_irqrestore() method, 185-187

spins, 184

stable kernel, 8-10

maintenance, 403

stacks

interrupt context, 122-123

interrupt stacks, 122

statically allocating memory on,
252-253

statements, switch statements, coding style,
396-397

statically allocating memory on stack,
252-253

statistics, debugging, 374

Stevens, W. Richard, 409

storing process descriptors, 26-27

structure padding, data alignment, 387-389

strcpy() function, 5

STREAMS, 8

structure initializers, coding style, 402-403

submitting

bug reports, 403-404

patches, 406

subscribing to Linux Kernel Mailing List
(LKML), 395

superblock data structure, 264

superblock object, VFS (Virtual Filesystem),
265-269

Swift, Jonathan, 390

switch statements, coding style, 396-397

switch_mm() method, 380

switch_to() method, 380

symmetrical multiprocessing

concurrency, 167

introduction of, 161-162

symmetrical multiprocessor (SMP), 8

synchronization, 162-168, 172

kernel, 21

reasons, 162-163

synchronization methods, 175

atomic operations, 175

64-bit operations, 180-181

bitwise operations, 181-183

converting, 177

counter implementation, 177

defining, 177

increments, 175-176

- integer operations, 176-179
- interfaces, 176
- nonatomic bit operations, 183
- overhead advantages, 179
- testing, 177
- barriers, 203-206
- BKL (Big Kernel Lock), 198-199
- completion variables, 197-198
- mutexes, 195-197
- ordering, 203-206
- preemption disabling, 201-202
- semaphores, 190-191
 - binary semaphores, 191-192
 - counting semaphores, 191-192
 - creating, 192-193
 - implementing, 193-194
 - initializing, 192
 - reader-writer semaphores, 194-195
- sequential locks, 200-201
- spin locks, 183-186
 - bottom halves, 187-188
 - reader-writer spin locks, 188-190
- synchronous interrupts, 114**
- syscalls. See system calls**
- sysfs, 337**
- sysfs filesystem, 355**
 - adding and removing kobjects, 357-358
 - adding files, 358-361
 - dentries, 355
 - Kernel Event Layer, 361-362
 - root directories, 357
- syslogd daemon, kernel messages, 367**
- SysRq commands, 371**
- system call() function, 73**
- system calls, 5, 69**
 - accessing, 71
 - accessing from user-space, 81-82
 - alternatives, 82-83
 - API (Application Programming Interface), 70
 - arguments, 71
 - behaviors, 71-72
 - binding, 79-81
 - C library, 70-71
 - clone(), 32
 - context, 78-81
 - denoting correct calls, 73
 - handlers, 73-74
 - implementation, 74-78
 - kernel, 71
 - multiplexing, 74
 - numbers, 72
 - parameter passing, 74
 - performance, 72
 - POSIX, 70
 - process scheduler, 65-67
 - processor affinity, 66
 - processor time, yielding, 66
 - pros and cons, 82
 - purpose of, 69
 - return values, 71
 - scheduler, 65-66
 - sched_getaffinity(), 66
 - sched_getscheduler(), 66
 - sched_get_priority_max(), 66
 - sched_setaffinity(), 66
 - sched_setparam(), 66
 - sched_setscheduler(), 66
 - sched_yield(), 67
 - side effects, 71
 - verifying, 75-78
- system timers, 207-208, 217**
- system uptime, 207-208**

T

Tanenbaum, Andrew, 407

tarball

- installing, 12
- source code, 11

task lists, 24-25

task queues, bottom halves, 135

TASK_INTERRUPTIBLE process, 27

TASK_RUNNING process, 27

TASK_STOPPED process, 28

task_struct, 24

TASK_TRACED process, 28

TASK_UNINTERRUPTIBLE process, 28

tasklet action() function, 143

tasklet disable() function, 145

tasklet disable nosync() function, 145

tasklet enable() function, 146

tasklet handlers, writing, 145

tasklet hi action() function, 143

tasklet hi schedule() function, 143

tasklet kill() function, 146

tasklet schedule() function, 143

tasklets, 137

- BH interface, 148
- bottom half mechanism, 142-143
- bottom halves, 136
- concurrency, 167
- declaring, 144-145
- implementing, 142-144
- ksoftirqd threads, 146-147
- scheduling, 143-146
- softirq types, 140
- structure, 142

TASKLET_SOFTIRQ tasklet, 140

tasks, 24

- ksoftirqd, 35
- parentless tasks, 38-40

pdflush, 35

sleeping, 58-61

waking up, 61

temporal locality, 323

temporary high memory mappings, 254-255

terminating processes, 36-40

testing atomic operations, 177

text section (processes), 23

Thompson, Ken, 1, 3

thrashing the cache per-CPU data, 258

thread support, Linux, 8

thread_info structure, 26

threads, 23, 34, 305

avoiding congestion, 334-335

bdflush, 333-334

concurrent programming, 33

contended, 184

creating, 34

deadlocks, 169-171

flusher threads, 331-335

kernel, 35-36

ksoftirqd, 146-147

kupdated, 333-334

lightweight processes, 34

Linux implementation, 33-36

migration threads, 66

parallelism, 33

pdflush, 333-334

worker threads, 149

threads of execution, 23

critical regions, 162

defined, 161

race conditions, 162

tick rate, Hz (hertz), 208-212

tick_periodic() function, 217, 219-220

tickless operating system, 212

time

- absolute time, 207
- current date and time, 220-221
- HZ, 391
- importance of, 207
- kernel's concept of, 208
- relative time, 207

time accounting, process scheduler, 50-52

time complexity, algorithms, 110-111

time stamp counter (TSC), 217

time() system call, 221

timeouts, wait queues, sleeping on, 229

timer interrupt, 207-208

timer interrupt handler, 217-220

TIMER_SOFTIRQ tasklet, 140

timers

- busy looping, 225-226
- delaying execution, 225-230
- deleting, 223
- dynamic timers, 207, 222
- hitting, 208
- implementation, 224-230
- internal values, 222
- interrupt handler, 217-220
- interrupts, frequencies, 209
- kernel, 136
- modifying, 223
- popping, 208
- popularity of, 222
- purpose of, 222
- race conditions, 224
- small delays, 226-227
- system timer, 217
- using, 222-223

timeslice count, processes, 211

timeslices

- process scheduler, 42
- process scheduler policy, 45

timespec data structure, 220

tools directory, kernel source code, 13

top halves, interrupt handlers, 115, 133

Torvalds, Linus, 3

transposition, printk() function, 367

traversing linked lists, 93-96

trees

adding processes to, 54-55

removing processes from, 56-57

VMA's (virtual memory areas), 313-314

tristates, 14

Tru64 (Digital), 2

true concurrency, 167

try to wake up() function, 61

two-list strategy, cache eviction, 325-326

type flags, 241-242

typedefs, coding style, 401

U

udelay() function, 227

UIDs (unique identification numbers), 100

allocating, 101-102

looking up, 102

removing, 103

uid_t data type, 384

***Understanding Linux Network Internals*, 408**

**University of California at Berkeley, BSD
(Berkeley Software Distributions), 2**

Unix, 1

characteristics, 2-3

creators, 1

development history, 1-2

evolution, 3

filesystems, 263-264

Linux, compared, 6-8

popularity of, 1

***Unix Internals: The New Frontiers*, 408**

Unix systems, scheduling, 47-48

*UNIX Systems for Modern Architectures:
Symmetric Multiprocessing and Caching*, 408

unlikely() function, 20

unlock kernel() function, 199

up() function, 193-194

update_curr() functions, 51-52

update_process_times() function, 218, 224

update_wall_time() function, 218

upping semaphores, 192

user preemption, process scheduler, 62-63

user spaces, jiffies global variable, 216

user-space, 5

- accessing system calls, 81-82

usr directory, kernel source tree, 13

utilities, diffstat, 405

V

Vahalia, Uresh, 408

van der Linden, Peter, 409

variables

- completion variables, 197-198
- condition variables, debugging, 374
- global variables, jiffies, 212-216
- xtime, 220

vendor kernels, 14

verb string, Kernel Event Layer, 361

vfork() function, 33-34

VFS (Virtual Filesystem), 261

- data structures, 265-266, 285-286
 - processes, 286-288
- file system type structure, 266
- interface, 261-262
- Linux filesystems, 288
- objects, 265-266
 - dentry, 265, 275-279
 - directory, 265
 - file, 265, 279-284
 - inode, 265, 270-274
 - operations, 265
 - superblock, 265-269
- vfsmount structure, 285-286
- virt directory, kernel source code, 13
- virtual device drivers, 338
- Virtual Filesystem (VFS)
 - dentry object, 275, 278
 - file object, 282
 - inode object, 270-272
 - superblock object, 267
 - vfsmount structure, 266
- Virtual Filesystem (VFS). *See* VFS (Virtual Filesystem)
- virtual memory, VMAs (virtual memory areas), 309-310, 314-315
 - flags, 311-312
 - lists, 313-314
 - operations, 312-313
 - private mapping, 312
 - shared mapping, 312
 - trees, 313-314
- virtual runtime, processes, 51-52
- virtual-to-physical address lookup, 321
- vmalloc() function, 244-245, 259
- VMAs (virtual memory areas), 309-310, 314-315
 - flags, 311-312
 - lists, 313-314
 - locating, 316-317
 - operations, 312-313
 - private mapping, 312
 - shared mapping, 312
 - trees, 313-314
- void local bh disable() function, 158
- void local bh enable() function, 158
- voluntary locks, 166

VSF

- abstraction layer, 262-263
- UNIX filesystems, 263-264

W-X-Y

- wait for completion() function, 198**
- wait queues, 58-59**
 - sleeping on, 229
- wait() function, 24**
- wake up() function, 61**
- wake_up_process() function, 36**
- websites, Linux Kernel Mailing List (LKML), 395**
- Windows Internals: Covering Windows Server 2008 and Windows Vista, 408***
- wmb() function, 204-205**
- word size, 381-384**
 - characters, 381
 - doublewords, 382
 - pages, 381
 - usage rules, 383
- work queue handler, 153**
- work queues, 137, 151**
 - bottom half mechanism, 149, 153
 - old task queues, 155-156
 - queue creation, 154-155
 - relationships among data structures, 152-153
 - run_workqueue() function, 151-152
 - thread data structure, 149
 - thread data structures, 150-151
 - work creation, 153
 - work flushing, 154
 - work scheduling, 153
- creating, 154-155
- implementing, 149-153
- scheduling, 153-154

- worker thread() function, 151**
- worker threads, 149**
- wraparounds, jiffies global variables, 214-216**
- wrapping interfaces, 402**
- write caching, 324**
- write lock irq() function, 189**
- write lock irqsave() function, 189**
- write lock() function, 189**
- write trylock() function, 190**
- write unlock irq() function, 189**
- write unlock irqrestore() function, 190**
- write unlock() function, 189**
- write() function, 5**
- write() system call, 261**
- write-through caches, 324**
- writepage() method, 328**
- writes starving reads, 300**
- writing**
 - interrupt handler, 118-119
 - tasklet handlers, 145

xtime variable, 220-221

yield() system call, 67

yielding

- process scheduler, 42
- processor time, 66

Z

- zeroed pages, obtaining, 236-237**
- zone modifiers, gfp_mask flags, 240**
- zones, 234**
 - pages, 233-235
 - ZONE_DMA, 233-235
 - ZONE_DMA32, 233
 - ZONE_HIGHMEM, 233
 - ZONE_NORMAL, 233