

经典图书新版

HZ BOOKS
华章科技

PEARSON

学习Linux内核最佳读物



Linux 内核设计 与实现 (英文版·第3版)

Linux Kernel Development (Third Edition)

(美) Robert Love 著



机械工业出版社
China Machine Press



Linux内核设计 与实现

Linux Kernel Development (Third Edition)

(美) Robert Love 著



机械工业出版社
China Machine Press

English reprint edition copyright © 2011 by Pearson Education Asia Limited and China Machine Press.

Original English language title: *Linux Kernel Development, Third Edition* (ISBN 978-0-672-32946-3) by Robert Love, Copyright © 2010.

All rights reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Addison-Wesley.

For sale and distribution in the People's Republic of China exclusively (except Taiwan, Hong Kong SAR and Macau SAR).

本书英文影印版由Pearson Education Asia Ltd. 授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

仅限于中华人民共和国境内（不包括中国香港、澳门特别行政区和中国台湾地区）销售发行。

本书封面贴有Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

封底无防伪标均为盗版

版权所有，侵权必究

本书法律顾问 北京市展达律师事务所

本书版权登记号：图字：01-2010-7526

图书在版编目（CIP）数据

Linux内核设计与实现（英文版·第3版）/（美）洛夫（Love, R.）著. —北京：机械工业出版社，2011.1

书名原文：Linux Kernel Development, Third Edition

ISBN 978-7-111-32792-9

I. L… II. 洛… III. Linux操作系统—程序设计—英文 IV. TP316.89

中国版本图书馆CIP数据核字（2010）第246255号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：迟振春

北京京北印刷有限公司印刷

2011年1月第1版第1次印刷

170mm × 242mm • 29印张

标准书号：ISBN 978-7-111-32792-9

定价：69.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88378991；88361066

购书热线：（010）68326294；88379649；68995259

投稿热线：（010）88379604

读者信箱：hzjsj@hzbook.com

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 time-less 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.

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

xiv Contents

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