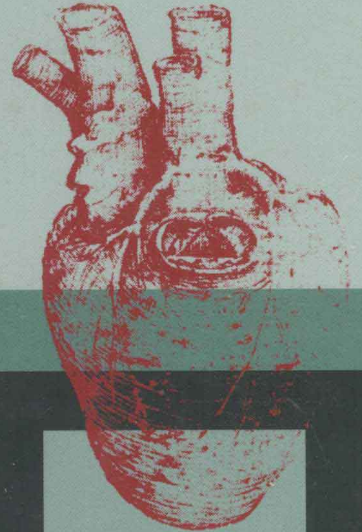
OPERATING SYSTEM
SOURCE CODE SECRETS VOLUME 1

The Basic Kernel Source Code Secrets™

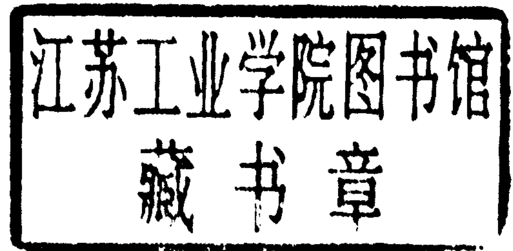
The most comprehensive operating system series ever! This "Gray's Anatomy" is a must for understanding where UNIX, Windows NT, Mach, and other modern systems will head in the next decade.



William Frederick Jolitz and
Lynne Greer Jolitz

Source Code Secrets
The Basic Kernel

Lynne Greer Jolitz
and William Frederick Jolitz



Peer-to-Peer Communications
San Jose, California

Operating System Source Code Secrets
Volume 1: The Basic Kernel
Copyright ©1996 William and Lynne Jolitz. All rights reserved.

"Source Code Secrets" is a trademark of William and Lynne Jolitz.
Some code discussed in this book is copyright ©1989, 1991-1993
the Regents of the University of California. All rights reserved. This
software is based in part on BSD Networking Software, Release 2
licensed from the Regents of the University of California.

Published by Peer-to-Peer Communications, Inc.
P.O. Box 640218
San Jose California 95164-0218, U.S.A.
Phone: 800-420-2677
Fax: 408-435-0895
E-mail: info@peer-to-peer.com

Cover Design: Susan Wilson, Palo Alto, CA
Production: Barclay Press, Newberg, OR
Printing: Paramount Graphics, Beaverton, OR

Printed in the United States of America
2 3 4 5 6 7 8 9 10

ISBN 1-57398-026-9

Peer-to-Peer offers discounts on this book when ordered in bulk
quantities. For more information, contact the Sales Deptartment
at the address above.

For a catalog of Peer-to-Peer Communications' other titles,
please contact the publisher.

This book is
Volume I

of

Operating System Source Code Secrets

東京都立短期大学

试读结束，需要全本PDF请购买 www.ertongbook.com

This Book is Dedicated to
William Leonard Jolitz (dec.)

An engineer whose knowledge
and dreams have traveled
beyond the solar system.

FOREWORD

An operating system is more than just a set of simple running modules—it is actually a complicated mechanism. The *heart* of an operating system is the basic kernel. Like a human heart, the basic kernel provides a complex operating system a fundamental means to distribute resources to all other subsystems in order for them to function independently. Without the basic kernel, no other portion of the operating system can "live." The virtual memory system, on the other hand, is the *brain* of an operating system, since it is through the virtual memory system that temporary and permanent memory is stored and accessed. Like a human brain, temporary memory is used as a workspace to assemble a new item before it is stored into permanent memory.

The modules and subsystems of the kernel, like biological ones, are also *interdependent.* Even "illnesses" which arise in one module and subsystem can manifest themselves in another. For example, in biological systems, if blood circulation is inadequate, the problem can appear as a neurologic disorder (i.e. dizziness or blurred vision). Likewise, if in the basic kernel (heart) interrupts are blocked for too long, the networking software (the nervous system) will not be able to acknowledge fast enough, resulting in service which is inefficient or fails. If the kernel memory allocator in the basic kernel has been misdesigned so that it suffers memory "leaks" (a common problem in commercial operating systems), the virtual memory system (brain) will suffer memory "loss," eventually resulting in system failure. As such, a thorough understanding of the *entire* kernel is crucial to proper operating system design.

Other modules and systems in the kernel have similar analogs in biological systems. The filesystem forms the *skeletal* structure of the operating system. Sockets, acting as conduits for information, outline a nervous system structure for the operating system as a whole, while the Internet networking protocols act as neurotransmitters for this information. The combination of these two modules complete the central nervous system for the operating system.

A biological analogy to operating systems design has its limits, since the primary motivation of biological systems is to survive and replicate. Neither of these functions can yet be done by any operating system, since our knowledge of control dynamics is

still in its infancy.[1] The point of this analogy is not to achieve complete similarity but instead to foster an understanding of the roles, responsibilities, and interdependencies of the portions of the kernel, just as an anatomy text organizes the study of the human body as a means for exposition. With this series we undertake to organize a complete understanding of an operating system kernel in order to give the reader the ability to evaluate and evolve any operating system.

[1] Error recovery and clustering are two simple mechanisms discussed throughout this series that attempt to replicate a portion of these functions, but these are still primitive means compared with the elegance of biological systems.

PREFACE

Obtaining an understanding of the core operating system kernel is not as simple as might appear—even if one has access to source code. Real world operating systems, unlike their academic paper counterparts, are trendy beasts, loaded down with mounds of additional hacks, functions, files, and other superfluous items with no regard for design, flexibility, extensibility, or future technology needs. The only thing that matters (and the only thing that a team of overloaded programmers have time for) is responding to customer demands in a timely manner.

Wading through all of this additional code to find the gems is a daunting task. However, there is a basic structure and rational design at the heart of even the most bloated operating system, but, like an archeologist searching for that elusive fossil, it may be buried so deeply that one could spend a lifetime trying to find it.

The **386BSD Operating System Reference Series** describes in detail the design and implementation decisions affecting key kernel source files made during the development of an operational kernel—386BSD. By discussing these basic components in detail and cross-referencing them to other components in the 386BSD kernel, the interwoven and complex structure of the kernel begins to take shape in a form which emphasizes its minimalist roots in a comprehensible manner. Through a rigorous discussion of these kernel files, the strengths and flaws of the various components of each file are revealed.

This agonizing reappraisal directly results in either new design or redesign (some of which appears in 386BSD Release 1.0), or a strengthened justification for the current design based on meticulous scrutiny—not assumption. Finally, a thorough understanding of current implementation feeds directly back into an understanding of the basic *levels of abstraction* which describe the kernel at its most elemental.[1]

Future direction and specific implementation suggestions to the reader are discussed in detail for these files. By gaining insight into the past, present, and future directions of 386BSD development, the reader now has enough information to explore the kernel

[1] See Chapter 2, **386BSD From the Inside-Out** (due 1997), for a detailed discussion of kernel level of abstraction.

further by working independently towards the future and not wasting time reinventing or reincorporating obsolete code from the past.

Among the volumes already written or in progress in this reference series are: **The Basic Kernel, The Virtual Memory Subsystem, The Filesystem, POSIX and Windows Sockets, The Internet Protocol Suite**, and **The UFS Filesystem**. It is from the basic files and subsystems discussed in these volumes that a modern operating system is made.

Isn't Source Code Enough?

The question which begs at this point is typically something along the lines of "Why should I bother reading about how the system kernel is constructed and designed when I already have the source code and an operational system to play with—after all, don't I already have everything I need to understand, fix, and modify it?" In particular, the wealth of 386BSD source code (including utilities and applications programs) must appear to answer all the dreams of the user previously denied any access to source code and system software.

Although the authors have been working with operating systems source code for a fair number of years (count in the decades), even we find new surprises with a kernel which has changed over 20 years of development and has had literally thousands of man-years put into it. As such, while having the source code is a powerful thing, it is not nearly enough to provide an overview of its many designers' intentions. If this were true, for example, Landau's landmark book on **Classical Mechanics**[2] would provide all the information required for the beginning physics student to derive all classical mechanics formulas (including Hamiltonian equations) with no further need for additional texts, lectures, or other information. While there are probably a few genius physicists out there who might claim to be able to do just this, the vast majority generally require a greater degree of insight from a variety of sources. In sum, it is a difficult area to comprehend immediately—even if you are a genius.

It is no longer enough to possess an operational operating system. Instead, it is imperative that its details be documented, examined and explained so that rational revision and extension is possible. Otherwise, it becomes difficult to determine if an operating system is correctly architected and implemented because the arcane details overwhelm the structure and obliterate the fine points of its design. In those cases, it may be impossible to tell the difference between an improvement in an artifact of the implementation and an improvement in a fundamental operating system paradigm shift. Indeed, the source code may even unintentionally obfuscate the paradigm beyond recognition, if it were not for other "sources" of information on the given system.

[2] L. D. Landau, & E.M. Liftshitz, **Mechanics**, Pergamon Press, Oxford (1960).

试读结束，需要全本PDF请购买 www.ertongbook.com

A modern operating system is far more than just a body of code—it must have well-justified reasons for being. If the rationale behind it is not firm, probably neither is the implementation. As such, both may require considerable revision to better approximate the ideal case.

Structure of This Book

Source Code Secrets Volume 1 examines the basic kernel source files required for a minimalist kernel design. Every operating system has a "simplest" model of form and structure, and many of them use similar mechanisms to deal with similar requirements. The primitive mechanics of the kernel and details of the kernel program as implemented on a target computer processor must be described in order to adapt the rest of the kernel program to that processor. The kernel itself is a program that benefits from a set of services that are used to make the program easier to manage and more concise. Not only must the kernel run a single application program, but it must allow for many application programs to be run concurrently through the concept of a process. And finally, the interface to each of these processes must reflect the POSIX standard operating system interface which, in this implementation, is directly implemented by the kernel.

This book is divided into five sections: **Introduction to the Basic Kernel (Chapter 1), The Machine-Dependent Kernel** (**Chapters 2** and **3**), **Internal Kernel Services (Chapter 4), The Process Object** (**Chapters 5, 6, 7, and 8**), and **POSIX Operating System Functionality** (**Chapter 9**).

An instructor's guide containing course outline and lecture notes for this entire series is in progress. Contact the publisher for further information.

How to Use This Book

Since this book describes the actual implementation of a working basic operating system kernel, it is intended to be used in tandem with an operating systems structure and design book such as **386BSD From the Inside-Out**. Examples and laboratory assignments should:

- Make concrete the concepts presented in each chapter.
- Illustrate the competing choices and the cost trade-offs implied.
- Clearly delimit how far the implementation may be extended.

In addition, code should be examined from the very beginning, from chapter to chapter, as we see the basic kernel built from the lowest level of abstraction layers of the basic kernel to the highest levels.

386BSD Source Code and Other Information

In 1989, 386BSD was begun by the authors as a means to encourage new ideas in operating systems and networking design and has gone through a number of major and minor releases. The port of BSD to the 386 PC was chronicled in a 17-part feature series published in **Dr. Dobbs Journal** called *Porting UNIX to the 386.*[3] This series is still one of the best references to understanding the genesis of 386BSD, as well as applying the methodology of an operating systems port to a new architecture.

The complete 386BSD operating system is now available as a bootable Window/UNIX CD-ROM containing over 574 Mbytes of source and binary. It also contains the complete text of the article series as will as other hyperlinked information on 386BSD by the authors in Windows help files. The **386BSD Reference CD-ROM** can be ordered using the order form at the very back of the book. It is strongly recommended for any serious study of operating systems. In addition, up-to-date information on 386BSD can be had by examining the 386BSD web site http://www.386bsd.org/

Acknowledgements

386BSD has been a continuing work since its genesis in 1989 by the authors, and while it might appear complete, there is still much more to explore. The basis work of 386BSD actually owes its lineage to the talents of a great many people over the last thirty years, from the contributions of those who worked on prior operating systems projects such as CTSS, the Berkeley TimeSharing System, MULTICS, UNIX, MACH, and of course Berkeley UNIX, coupled with more recent contributions (primarily in the areas of utilities and applications) by people on the Internet.

This current project would not have been possible without the help and encouragement of many people who wanted to see this legacy continued into the future. We would especially like to thank Thos Sumner for his critical review and suggestions while we wrote this book. We would also like to thank Peter Hutchinson, Manny Sawit, and Stan Barnes of Miller-Freeman for their unstinting support of this documentation project, and Jon Erickson and Ray Valdéz of **Dr. Dobbs Journal** for their support of our early porting efforts (resulting in an article series). Finally, we would like to thank Paul Fronberg for his encouragement in talking about and teaching this material.

In the production of this book and formulation of the rest of this series, we would like to thank Dan Doernberg and Rachel Unkefer, Chris MacIntosh (production coordination), Darwin Melnyk and Darren Gilroy (production), and Susan Wilson (cover art), Daniel Hobbs (proofing and word list), and Hank Kennedy (contracts).

[3] January through November 1991 and February through July 1992. Follow-on articles on 386BSD have since appeared in this magazine as well.

Perspective: 386BSD and the "Real World"

The perspective of these writings is a combined set of views reflecting stability, standards compatibility, raw "bit-level" performance, and extensibility/scalability. No attempt is made to extend this perspective to other areas, such as providing compatibility with arbitrary commercial products, nor are testing or support isolation a portion of this work since this is not intended as a commercial quality operating system. While the quality of most commercial systems may be in decline, this should not be viewed as a permanent condition but instead one that will be eventually remedied by market forces when the customer has become sufficiently educated to see through the smoke and haze. As a result, providing enough perspective to see through the marketing smokescreen to the actual work which underlies it is another key portion of this work (something which may not necessarily be appreciated by those proffering commercial systems).

Acuity is thus a deft tool to separate operating system fact from fancy. We hope our work here helps you to hone your edge—if just a bit sharper. If so, please use it in a positive and productive direction.

> Finally: It was stated at the outset, that this system would not be here, and at once, perfected. You cannot but plainly see that I have kept my word. But I now leave my cetological System standing thus unfinished, even as the great Cathedral of Cologne was left, with the crane still standing upon the top of the uncompleted tower. For small erections may be finished by their first architects; grand ones, true ones, ever leave the copestone to posterity. God keep me from ever completing anything. This whole book is but a draught - nay, but the draught of a draught. Oh, Time, Strength, Cash, and Patience!
>
> —Herman Melville, *Moby Dick*, Chapter 32

CONTENTS

FOREWORD 7

PREFACE 9

Isn't Source Code Enough? 10
Structure of This Book 11
How to Use This Book 11
386BSD Source Code and Other Information 12
Acknowledgements 12
Perspective: 386BSD and the "Real World" 13

CONTENTS 15

INTRODUCTION TO THE BASIC KERNEL 23

The Machine-Dependent Kernel 24
The Internal Kernel Services 24
The Process Object 25
POSIX Operating System Functionality 26
Structure of the Kernel Program 26
Structure of the Annotations 26
Perspective: Why Write Annotations? 27

ASSEMBLY ENTRY AND PRIMITIVES (i386/locore.s) 29

Locore.s Development Decisions 29
- Historical Origins 30
- System Initialization and Operation 30
- The Context Switch Mechanism 31
- Entry to / Exit from the Kernel 31

Kernel Program as Loaded by the Bootstrap 32
Kernel Initial Memory Map 33
locore.s Functions and Terminology 34
- What is start? 35
- What is reloc? 51
- What is rfillkpt? 51
- What is fillkpt? 52
- What is icode? 53

What is sigcode? 55
What is ssdtosd? 56
What is copyout? 57
What are copyout_4, copyout_2, and copyout_1? 62
What is copyoutstr? 65
What is swtch? 68
What is qswtch? 72
What is idle? 73
What are Processor Exceptions? 74
How are Processor Exceptions Implemented? 74
386BSD Processor Exception Design Choices and Trade-Offs 77
What are Peripheral Device Interrupts? 77
How are Peripheral Device Interrupts Implemented? 78
What are Set Processor Level Functions? 90
How are Set Processor Level Functions Implemented? 92
What are System Call 'Call Gates'? 96
How are System Call Gates Implemented? 97

CPU-SPECIFIC PRIMITIVES (i386/trap.c, i386/cpu.c) 101
Processor Exception and System Call Entry Handling: i386/trap.c 102
Functions Contained in the File i386/trap.c 102
What is trap()? 103
What is trapexcept()? 119
What is trapwrite()? 121
What is copyin3()? 122
What is syscall()? 124
What is systrap()? 131
CPU Kernel Facilities: i386/cpu.c 133
Threads versus Processes 133
POSIX Signals 133
Functions Contained in the File i386/cpu.c 134
What is cpu_tfork? 134
What is cpu_texit ? 142
What is cpu_execsetregs()? 145
What is cpu_signal()? 146
What is cpu_signalreturn()? 150
What is cpu_reset()? 153
What is cpu_ptracereg()? 154

INTERNAL KERNEL SERVICES (kern/config.c, kern/malloc.c) 157
Issues in Configuration 157
PC Device Conflicts 158
The Built-in Obsolescence of PC Static Configuration Mechanisms 159
Plug and Play: A Growing Necessity 160
Configuration in 386BSD: kern/config.c 161

试读结束，需要全本PDF请购买 www.ertongbook.com