

THE **DESIGN AND IMPLEMENTATION** OF THE FreeBSD OPERATING SYSTEM

SECOND EDITION

MARSHALL KIRK MCKUSICK GEORGE V. NEVILLE-NEIL **•** ROBERT N.M. WATSON

FREE SAMPLE CHAPTER







The Design and Implementation of the

FreeBSD_® Operating System

Second Edition

This page intentionally left blank

The Design and Implementation of the **FreeBSD**® **Operating System**

Second Edition

Marshall Kirk McKusick

George V. Neville-Neil

Robert N.M. Watson

♣Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco New York • Toronto • Montreal • London • Munich • Paris • Madrid Capetown • Sydney • Tokyo • Singapore • Mexico City UNIX is a registered trademark of X/Open in the United States and other countries. FreeBSD and the FreeBSD logo used on the cover of this book are registered and unregistered trademarks of the FreeBSD Foundation and are used by Pearson Education with the permission of the FreeBSD Foundation. 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 Pearson was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors 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.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the United States, please contact international@pearsoned.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

McKusick, Marshall Kirk.
The design and implementation of the FreeBSD operating system / Marshall Kirk McKusick, George V. Neville-Neil, Robert N. M. Watson.
pages cm
Includes bibliographical references and index.
ISBN-13: 978-0-321-96897-5 (hardcover : alk. paper)
ISBN-10: 0-321-96897-2 (hardcover : alk. paper)
I. FreeBSD. 2. Free computer software. 3. Operating systems (Computers)
I. Neville-Neil, George V. II. Watson, Robert N. M. III. Title.
QA76.774.F74M35 2014
005.4'32—dc23

Copyright © 2015 by 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. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-321-96897-5 ISBN-10: 0-321-96897-2 Text printed on recycled and acid-free paper at Courier in Westford, Massachusetts. Second Printing, November 2014

Dedication

This book is dedicated to the BSD community. Without the contributions of that community's members, there would be nothing about which to write. This page intentionally left blank

Contents

Preface	xxi
About the Authors	xxix
Part I Overview	1
Chapter 1 History and Goals	3
 1.1 History of the UNIX System 3 Origins 3 Research UNIX 4 AT&T UNIX System III and System V 5 Berkeley Software Distributions 6 UNIX in the World 7 	
1.2 BSD and Other Systems 7 The Influence of the User Community 8	
 1.3 The Transition of BSD to Open Source 9 Networking Release 2 10 The Lawsuit 11 4.4BSD 13 4.4BSD-Lite Release 2 13 	
1.4The FreeBSD Development Model14References17	
Chapter 2 Design Overview of FreeBSD	21
2.1 FreeBSD Facilities and the Kernel 21 The Kernel 22	
2.2 Kernel Organization 23	

Contents

2.3 Kernel Services 26 2.4 26 Process Management Signals 28 Process Groups and Sessions 29 2.5 Security 29 Process Credentials 31 Privilege Model 31 Discretionary Access Control 32 Capability Model 32 Jail Lightweight Virtualization 32 Mandatory Access Control 34 Event Auditing 35 Cryptography and Random-Number Generators 35 2.6 Memory Management 36 BSD Memory-Management Design Decisions 36 Memory Management Inside the Kernel 38 2.7 I/O System Overview 39 Descriptors and I/O 39 Descriptor Management 41 42 Devices Socket IPC 42 Scatter-Gather I/O 43 Multiple Filesystem Support 43 2.8 Devices 44 2.9 The Fast Filesystem 45 Filestores 48 2.10 The Zettabyte Filesystem 49 2.11 The Network Filesystem 50 2.12 Interprocess Communication 50 2.13 Network-Layer Protocols 51 2.14 Transport-Layer Protocols 52 2.15 System Startup and Shutdown 52 Exercises 54 54 References

Chapter 3 Kernel Services

3.1	Kernel Organization 57	
	System Processes 57	
	System Entry 58	
	Run-Time Organization 59	
	Entry to the Kernel 60	
	Return from the Kernel 61	
3.2	System Calls 62	
	Result Handling 62	
	Returning from a System Call	63
3.3	Traps and Interrupts 64	
	I/O Device Interrupts 64	

	Software Interrupts 65
3.4	Clock Interrupts 65
	Statistics and Process Scheduling 66
	Timeouts 67
3.5	Memory-Management Services 69
3.6	Timing Services 73
	Real Time 73
	External Representation 73
	Adjustment of the Time 74
	Interval Time 74
3.7	Resource Services 75
	Process Priorities 75
	Resource Utilization 75
	Resource Limits 76
	Filesystem Quotas 77
3.8	Kernel Tracing Facilities 77
	System-Call Tracing 77
	DTrace 78
	Kernel Tracing 82
	Exercises 84
	References 85

Part II Processes

Chapter 4 Process Management

4.1	Introduction to Process Management	89
	Multiprogramming 90	
	Scheduling 91	
42	Process State 92	
7.2	The Process Structure 94	
	The Thread Structure 08	
4.2	Customet Societalise 00	
4.3	Context Switching 99	
	Thread State 100	
	Low-Level Context Switching 100	
	Voluntary Context Switching 101	
	Synchronization 106	
	Mutex Synchronization 107	
	Mutex Interface 109	
	Lock Synchronization 110	
	Deadlock Prevention 112	
4.4	Thread Scheduling 114	
	The Low-Level Scheduler 114	
	Thread Run Queues and Context Switching	115
	Timeshare Thread Scheduling 117	
	Multiprocessor Scheduling 122	
	Adaptive Idle 125	
	10000110 125	

89

Traditional Timeshare Thread Scheduling 125

- 4.5 Process Creation 126
- 4.6 Process Termination 128
 4.7 Signals 129 Posting of a Signal 132 Delivering a Signal 135
 4.8 Process Groups and Sessions
- 4.8 Process Groups and Sessions 136 Process Groups 137 Sessions 138 Job Control 139
 4.9 Process Debugging 142 Exercises 144 References 146

Chapter 5 Security

5.1	Operating-System Security 148	
5.2	Security Model 149	
	Process Model 149	
	Discretionary and Mandatory Access Control	150
	Trusted Computing Base (TCB) 151	
	Other Kernel-Security Features 151	
5.3	Process Credentials 151	
	The Credential Structure 152	
	Credential Memory Model 153	
	Access-Control Checks 153	
5.4	Users and Groups 154	
	Setuid and Setgid Binaries 155	
5.5	Privilege Model 157	
	Implicit Privilege 157	
	Explicit Privilege 157	
5.6	Interprocess Access Control 159	
	Visibility 160	
	Signals 160	
	Scheduling Control 160	
	Waiting on Process Termination 161	
	Debugging 161	
5.7	Discretionary Access Control 161	
	The Virtual-Filesystem Interface and DAC	162
	Object Owners and Groups 163	
	UNIX Permissions 164	
	Access Control Lists (ACLs) 165	
	POSIX.1e Access Control Lists 168	
	NFSv4 Access Control Lists 171	
5.8	Capsicum Capability Model 174	
	Capsicum Application Structure 175	
	Capability Systems 176	
	Capabilities 177	

	Capability Mode 179
5.9	Jails 180
5.10	Mandatory Access-Control Framework 184
	Mandatory Policies 186
	Guiding Design Principles 187
	Architecture of the MAC Framework 188
	Framework Startup 189
	Policy Registration 190
	Framework Entry-Point Design Considerations 191
	Policy Entry-Point Considerations 192
	Kernel Service Entry-Point Invocation 193
	Policy Composition 194
	Object Labelling 195
	Label Life Cycle and Memory Management 196
	Label Synchronization 199
5 1 1	Policy-Agnostic Label Management from Userspace 199
5.11	Security Event Auditing 200
	Audit Events and Records 201 DSM Audit Decords and Audit Troils 202
	Kernel Audit Implementation 202
5 1 2	Cruptographic Services 206
5.12	Cryptographic Services 200
	Random-Number Generator 208
5 13	GELL Full Disk Encryption 212
5.15	Confidentiality and Integrity Protection 212
	Key Management 213
	Starting GELI 214
	Cryptographic Block Protection 215
	I/O Model 216
	Limitations 216
	Exercises 217
	References 217

Chapter 6 Memory Management

6.1 Terminology 221 Processes and Memory 222 223 Paging Replacement Algorithms 224 Working-Set Model 225 Swapping 225 Advantages of Virtual Memory 225 Hardware Requirements for Virtual Memory 226 Overview of the FreeBSD Virtual-Memory System 6.2 227 User Address-Space Management 228 Kernel Memory Management 6.3 230 Kernel Maps and Submaps 231 Kernel Address-Space Allocation 233 The Slab Allocator 236

The Keg Allocator 238 The Zone Allocator 239 Kernel Malloc 241 243 Kernel Zone Allocator 64 Per-Process Resources 244 FreeBSD Process Virtual-Address Space 245 Page-Fault Dispatch 245 Mapping to *Vm_objects* 247 Vm objects 249 Vm_objects to Pages 249 6.5 Shared Memory 250 Mmap Model 251 253 Shared Mapping Private Mapping 254 Collapsing of Shadow Chains 257 Private Snapshots 258 6.6 Creation of a New Process 258 Reserving Kernel Resources 259 Duplication of the User Address Space 260 Creation of a New Process Without Copying 261 6.7 Execution of a File 262 6.8 Process Manipulation of Its Address Space 263 Change of Process Size 263 File Mapping 264 Change of Protection 266 6.9 Termination of a Process 266 267 6.10 The Pager Interface Vnode Pager 269 Device Pager 270 Physical-Memory Pager 272 Swap Pager 272 6.11 Paging 276 Hardware-Cache Design 280 Hardware Memory Management 282 Superpages 284 6.12 Page Replacement 289 Paging Parameters 291 292 The Pageout Daemon Swapping 295 The Swap-In Process 296 6.13 Portability 298 The Role of the pmap Module 299 Initialization and Startup 301 Mapping Allocation and Deallocation 304 Change of Access and Wiring Attributes for Mappings 306 Maintenance of Physical Page-Usage Information 307 Initialization of Physical Pages 308 Management of Internal Data Structures 308

Contents

Exercises	308
References	310

Part III I/O System

313

315

Chapter 7 I/O System Overview

7.1	Descriptor Management and Services 316
	Open File Entries 318
	Management of Descriptors 319
	Asynchronous I/O 321
	File-Descriptor Locking 322
	Multiplexing I/O on Descriptors 324
	Implementation of <i>Select</i> 327
	Kqueues and Kevents 329
	Movement of Data Inside the Kernel 332
7.2	Local Interprocess Communication 333
	Semaphores 335
	Message Queues 337
	Shared Memory 338
7.3	The Virtual-Filesystem Interface 339
	Contents of a Vnode 339
	Vnode Operations 342
	Pathname Translation 342
	Exported Filesystem Services 343
7.4	Filesystem-Independent Services 344
	The Name Cache 346
	Buffer Management 347
	Implementation of Buffer Management 350
7.5	Stackable Filesystems 352
	Simple Filesystem Layers 354
	The Union Filesystem 355
	Other Filesystems 357
	Exercises 358
	References 359

Chapter 8 Devices

8.1	Device Overview 361	
	The PC I/O Architecture 362	
	The Structure of the FreeBSD Mass Storage I/O Subsystem	364
	Device Naming and Access 366	
8.2	I/O Mapping from User to Device 367	
	Device Drivers 368	
	I/O Queueing 369	
	Interrupt Handling 370	

xiii

431

8.3	Character Devices 370	
	Raw Devices and Physical I/O 372	
	Character-Oriented Devices 373	
	Entry Points for Character Device Drivers 373	
8.4	Disk Devices 374	
	Entry Points for Disk Device Drivers 374	
	Sorting of Disk I/O Requests 375	
	Disk Labels 376	
8.5	Network Devices 378	
	Entry Points for Network Drivers 378	
	Configuration and Control 379	
	Packet Reception 380	
	Packet Transmission 381	
8.6	Terminal Handling 382	
	Terminal-Processing Modes 383	
	User Interface 385	
	Process Groups, Sessions, and Terminal Control 387	
	Terminal Operations 388	
	Terminal Output (Upper Half) 388	
	Terminal Output (Lower Half) 389	
	Terminal Input 390	
	Closing of Terminal Devices 391	
8.7	The GEOM Layer 391	
	Terminology and Topology Rules 392	
	Changing Topology 393	
	Operation 396	
	Topological Flexibility 397	
8.8	The CAM Layer 399	
	The Path of a SCSI I/O Request Through the CAM Subsystem 400	
	ATA Disks 402	
8.9	Device Configuration 402	
	Device Identification 405	
	Autoconfiguration Data Structures 407	
0.10	Resource Management 412	
8.10	Device Virtualization 414	
	Interaction with the Hypervisor 414	
	Virtio 415	
	Xen 419	
	Every ass-fillougil 427	
	Exercises 428	
	kelerences 429	
Chapter 9 The Fast Filesystem		

9.1 Hierarchical Filesystem Management
9.2 Structure of an Inode 433 Changes to the Inode Format 435 Extended Attributes 436 New Filesystem Capabilities 438

File Flags 439 Dynamic Inodes 441 Inode Management 442 9.3 Naming 443 Directories 444 Finding of Names in Directories 446 Pathname Translation 447 449 Links 9.4 Ouotas 451 9.5 File Locking 454 96 459 Soft Updates Update Dependencies in the Filesystem 460 Dependency Structures 464 Bitmap Dependency Tracking 466 Inode Dependency Tracking 467 Direct-Block Dependency Tracking 469 Indirect-Block Dependency Tracking 470 Dependency Tracking for New Indirect Blocks 471 New Directory-Entry Dependency Tracking 472 New Directory Dependency Tracking 474 Directory-Entry Removal-Dependency Tracking 475 File Truncation 476 File and Directory Inode Reclamation 476 Directory-Entry Renaming Dependency Tracking 476 Fsync Requirements for Soft Updates 477 478 File-Removal Requirements for Soft Updates Soft-Updates Requirements for fsck 480 9.7 Filesystem Snapshots 480 Creating a Filesystem Snapshot 481 Maintaining a Filesystem Snapshot 483 Large Filesystem Snapshots 484 Background fsck 486 User-Visible Snapshots 487 Live Dumps 487 9.8 487 Journaled Soft Updates Background and Introduction 487 Compatibility with Other Implementations 488 Journal Format 488 Modifications That Require Journaling 489 Additional Requirements of Journaling 490 The Recovery Process 492 Performance 493 Future Work 494 Tracking File-Removal Dependencies 495 9.9 The Local Filestore 496 Overview of the Filestore 497 User I/O to a File 499 9.10 The Berkeley Fast Filesystem 501 Organization of the Berkeley Fast Filesystem 502

Boot Blocks 503 Optimization of Storage Utilization 504 Reading and Writing to a File 505 Layout Policies 507 Allocation Mechanisms 510 Block Clustering 514 Extent-Based Allocation 516 Exercises 517 References 519

Chapter 10 The Zettabyte Filesystem

- 10.1 Introduction 523 10.2 **ZFS** Organization 527 ZFS Dnode 528 ZFS Block Pointers 529 531 ZFS *objset* Structure 10.3 **ZFS Structure** 532 The MOS Layer 533 The Object-Set Layer 534 10.4 **ZFS** Operation 535 Writing New Data to Disk 536 Logging 538 RAIDZ 540 Snapshots 542 ZFS Block Allocation 542 Freeing Blocks 543
- Deduplication 545 Remote Replication 546 10.5 ZFS Design Tradeoffs 547 Exercises 549

References

Chapter 11 The Network Filesystem

549

11.1	Overview 551
11.2	Structure and Operation 553
	The FreeBSD NFS Implementation 558
	Client–Server Interactions 562
	Security Issues 564
	Techniques for Improving Performance 565
11.3	NFS Evolution 567
	Namespace 572
	Attributes 572
	Access Control Lists 574
	Caching, Delegation, and Callbacks 574
	Locking 581
	Security 583
	Crash Recovery 584

523

Exercises 586 References 587

Part	IV Interprocess Communication	591
Chap	ter 12 Interprocess Communication	593
12.1	Interprocess-Communication Model 593 Use of Sockets 596	
12.2 12.3	Implementation Structure and Overview599Memory Management601Mbufs601	
	Storage-Management Algorithms605Mbuf Utility Routines606	
12.4	IPC Data Structures606Socket Addresses611Locks612	
12.5	Connection Setup 612	
12.6	Data Transfer 615	
	Transmitting Data 616	
107	Receiving Data 617	
12./	Socket Shutdown 620 Natural Communication Protocol Internal Structure 621	
12.0	Data Flow 623 Communication Protocols 624	
12.9	Socket-to-Protocol Interface 626	
12.9	Protocol User-Request Routines 627	
	Protocol Control-Output Routine 630	
12.10	Protocol-to-Protocol Interface 631	
	pr_output 632	
	pr_input 632	
10.11	pr_ctlinput 633	
12.11	Protocol-to-Network Interface 634	
	Packet Transmission 641	
	Packet Reception 642	
12.12	Buffering and Flow Control 643	
	Protocol Buffering Policies 643	
	Queue Limiting 643	
12.13	Network Virtualization 644	
	Exercises 646	
	References 648	
Chap	ter 13 Network-Layer Protocols	649
13.1	Internet Protocol Version 4 650	
	IPv4 Addresses 652	
	Broadcast Addresses 653	

721

	Internet Multicast 654	
10.0	Link-Layer Address Resolution 655	
13.2	Internet Control Message Protocols (ICMP)	657
13.3	Internet Protocol Version 6 659	
	IPv6 Addresses 660	
	IPv6 Packet Formats 662	
	Changes to the Socket API 664	
	Autoconfiguration 666	
13.4	Internet Protocols Code Structure 670	
	Output 671	
	Input 673	
	Forwarding 674	
13.5	Routing 675	
	Kernel Routing Tables 677	
	Routing Lookup 680	
	Routing Redirects 683	
	Routing-Table Interface 683	
	User-Level Routing Policies 684	
	User-Level Routing Interface: Routing Socket	685
13.6	Raw Sockets 686	
	Control Blocks 686	
	Input Processing 687	
	Output Processing 687	
13.7	Security 688	
	IPSec Overview 689	
	Security Protocols 690	
	Key Management 693	
	IPSec Implementation 698	
13.8	Packet-Processing Frameworks 700	
	Berkeley Packet Filter 700	
	IP Firewalls 701	
	IPFW and Dummynet 702	
	Packet Filter (PF) 706	
	Netgraph 707	
	Netmap 712	
	Exercises 715	
	References 717	
Chap	ter 14 Transport-Layer Protocols	
1/1	Intermet Dorts and Associations 721	
14.1	Internet Ports and Associations /21	
14.0	Liser Determine Directored (UDD) 722	
14.2	User Datagram Protocol (UDP) /23	

Initialization 723 Output 724 Input 724 Control Operations 725 14.3 Transmission Control Protocol (TCP) 725 TCP Connection States 727

	Sequence Variables 730
14.4	TCP Algorithms 732
	Timers 733
	Estimation of Round-Trip Time 735
	Connection Establishment 736
	SYN Cache 739
	SYN Cookies 739
	Connection Shutdown 740
14.5	TCP Input Processing 741
14.6	TCP Output Processing 745
	Sending Data 746
	Avoidance of the Silly-Window Syndrome 746
	Avoidance of Small Packets 747
	Delayed Acknowledgments and Window Updates 748
	Selective Acknowledgment 749
	Retransmit State 751
	Slow Start 752
	Buffer and Window Sizing 754
	Avoidance of Congestion with Slow Start 755
	Fast Retransmission 756
	Modular Congestion Control 758
	The Vegas Algorithm 759
	The Cubic Algorithm 760
14.7	Stream Control Transmission Protocol (SCTP) 761
	Chunks 762
	Association Setup 762
	Data Transfer 764
	Association Shutdown 766
	Multihoming and Heartbeats 767
	Exercises 768
	References 770

Part V System Operation

Chapter 15 System Startup and Shutdown 775 15.1 Firmware and BIOSes 776 15.2 777 Boot Loaders Master Boot Record and Globally Unique Identifier Partition Table 778 The Second-Stage Boot Loader: gptboot 779 The Final-Stage Boot Loader: /boot/loader 779 Boot Loading on Embedded Platforms 781 15.3 Kernel Boot 782 Assembly-Language Startup 783 Platform-Specific C-Language Startup 784 Modular Kernel Design 785 Module Initialization 785

	Basic Kernel Services 787
	Kernel-Thread Initialization 792
	Device-Module Initialization 794
	Loadable Kernel Modules 796
15.4	User-Level Initialization 798
	/sbin/init 798
	System Startup Scripts 798
	/usr/libexec/getty 799
	/usr/bin/login 799
15.5	System Operation 800
	Kernel Configuration 800
	System Shutdown and Autoreboot 801
	System Debugging 802
	Passage of Information To and From the Kernel 803
	Exercises 805
	References 806

Glossary

Index

847

Preface

This book follows the earlier authoritative and full-length descriptions of the design and implementation of the 4.3BSD and 4.4BSD versions of the UNIX system developed at the University of California at Berkeley. Since the final Berkeley release in 1994, several groups have continued development of BSD. This book details FreeBSD, the system with the largest set of developers and the most widely distributed releases. Although the FreeBSD distribution includes nearly 1000 utility programs in its base system and nearly 25,000 optional utilities in its ports collection, this book concentrates almost exclusively on the kernel.

UNIX-like Systems

UNIX-like systems include the traditional vendor systems such as Solaris and HP-UX; the Linux-based distributions such as Red Hat, Debian, Suse, and Slackware; and the BSD-based distributions such as FreeBSD, NetBSD, OpenBSD, and Darwin. They run on computers ranging from smart phones to the largest supercomputers. They are the operating system of choice for most multiprocessor, graphics, and vector-processing systems, and are widely used for the original purpose of timesharing. The most common platform for providing network services (from FTP to WWW) on the Internet, they are collectively the most portable operating system ever developed. This portability is due partly to their implementation language, C [Kernighan & Ritchie, 1989] (which is itself a widely ported language), and partly to the elegant design of the system.

Since its inception in 1969 [Ritchie & Thompson, 1978], the UNIX system has developed in several divergent and rejoining streams. The original developers continued to advance the state of the art with their Ninth and Tenth Edition UNIX inside AT&T Bell Laboratories, and then their Plan 9 successor to UNIX. Meanwhile, AT&T licensed UNIX System V as a product before merging it with Sun Microsystem's BSD-based SunOS to produce Solaris. Ninth Edition UNIX, System V, and Solaris were all strongly influenced by the Berkeley Software Distributions produced by the Computer Systems Research Group (CSRG) of the University of California at Berkeley. The Linux operating system, although developed independently of the other UNIX variants, implements the UNIX interface. Thus, applications developed to run on other UNIX-based platforms can be easily ported to run on Linux.

Berkeley Software Distributions

The distributions from Berkeley were the first UNIX-based systems to introduce many important features including the following:

- Demand-paged virtual-memory support
- Automatic configuration of the hardware and I/O system
- A fast and recoverable filesystem
- The socket-based interprocess-communication (IPC) primitives
- The reference implementation of TCP/IP

The Berkeley releases found their way into the UNIX systems of many vendors and were used internally by the development groups of many other vendors. The implementation of the TCP/IP networking protocol suite in 4.2BSD and 4.3BSD, and the availability of those systems, played a key role in making the TCP/IP networking protocol suite a world standard. Even the non-UNIX vendors such as Microsoft have adopted the Berkeley socket design in their Winsock IPC interface.

The BSD releases have also been a strong influence on the POSIX (IEEE Std 1003.1) operating-system interface standard, and on related standards. Several features—such as reliable signals, job control, multiple access groups per process, and the routines for directory operations—have been adapted from BSD for POSIX.

Early BSD releases contained licensed UNIX code, thus requiring recipients to have an AT&T source license to be able to obtain and use BSD. In 1988, Berkeley separated its distribution into AT&T-licensed and freely redistributable code. The freely redistributable code was licensed separately and could be obtained, used, and redistributed by anyone. The final freely redistributable 4.4BSD-Lite2 release from Berkeley in 1994 contained nearly the entire kernel and all the important libraries and utilities.

Two groups, NetBSD and FreeBSD, sprang up in 1993 to begin supporting and distributing systems built from the freely redistributable releases being done by Berkeley. The NetBSD group emphasized portability and the minimalist approach, porting the systems to nearly 60 platforms and they were determined to keep the system lean to aid embedded applications. The FreeBSD group emphasized maximal support for the PC architecture and pushed to ease installation for, and market their system to, as wide an audience as possible.

In 1995, the OpenBSD group split from the NetBSD group to develop a distribution that emphasized security. In 2003, the Dragonfly group split from the FreeBSD group to develop a distribution that used a significantly lighter-weight mechanism to support multiprocessing. Over the years, there has been a healthy

xxii

competition among the BSD distributions, with many ideas and much code flowing between them.

Material Covered in this Book

This book is about the internal structure of the FreeBSD 11 kernel and about the concepts, data structures, and algorithms used in implementing FreeBSD's system facilities. The book covers FreeBSD from the system-call level down—from the interface to the kernel to the hardware itself. The kernel includes system facilities, such as process management, security, virtual memory, the I/O system, filesystems, the *socket* IPC mechanism, and network protocol implementations. Material above the system-call level—such as libraries, shells, commands, programming languages, and other user interfaces—is excluded, except for some material related to the terminal interface and to system startup. Following the organization first established by Organick's book about Multics [Organick, 1975], this book is an in-depth study of a contemporary operating system.

Where particular hardware is relevant, the book refers to the Intel 32-bit architecture and the similar AMD 64-bit architecture. Because FreeBSD has emphasized development on these architectures, they are the architectures with the most complete support and so provide a convenient point of reference.

Use by Computer Professionals

FreeBSD is widely used to support the core infrastructure of many companies worldwide. Because it can be built with a small footprint, it is also seeing increased use in embedded applications. The licensing terms of FreeBSD do not require the distribution of changes and enhancements to the system. The licensing terms of Linux require that all changes and enhancements to the kernel be made available in source form at minimal cost. Thus, companies that need to control the distribution of their intellectual property build their products using FreeBSD.

This book is of direct use to the professionals who work with FreeBSD systems. Individuals involved in technical and sales support can learn the capabilities and limitations of the system; applications developers can learn how to interface with the system effectively and efficiently; system administrators without direct experience with the FreeBSD kernel can learn how to maintain, tune, and configure the system; and systems programmers can learn how to extend, enhance, and interface with the system.

Readers who will benefit from this book include operating-system implementors, system programmers, UNIX application developers, administrators, and curious users. The book can be read as a companion to the source code of the system, falling as it does between the manual pages and the code in its level of detail. But this book is neither exclusively a UNIX programming manual nor a user tutorial. Familiarity with the use of some version of the UNIX system (see, for example, Stevens [1992]) and with the C programming language (see, for example, Kernighan & Ritchie [1989]) would be extremely useful. The FreeBSD Handbook gives a comprehensive introduction to the setting up, operation, and programming of FreeBSD [FreeBSD Mall, 2004; FreeBSD.org, 2014]. FreeBSD packaging, designed to be easy to install and use for both desktop and laptop users, is available in the PC-BSD distribution [Lavigne, 2010; PC-BSD.org, 2014].

Use in Courses on Operating Systems

This book is suitable for use as a reference text to provide background for a primary textbook in a first-level course on operating systems. It is not intended for use as an introductory operating-system textbook; the reader should have already encountered terminology such as "memory management," "process scheduling," and "I/O systems" [Silberschatz et al., 2012]. Familiarity with the concepts of network protocols [Comer, 2000; Stallings, 2000; Tanenbaum, 2010] will be useful for understanding some of the later chapters.

This book can be used in combination with a copy of the FreeBSD system for more advanced operating-systems courses. Students' assignments can include changes to, or replacements of, key system components such as the scheduler, the paging daemon, the filesystems, thread signalling, various networking layers, and I/O management. The ability to load, replace, and unload modules from a running kernel allows students to experiment without the need to compile and reboot the system. By working with a real operating system, students can directly measure and experience the effects of their changes. Because of the intense peer review and insistence on well-defined coding standards throughout its 35-year lifetime, the FreeBSD kernel is considerably cleaner, more modular, and thus easier to understand and modify than most software projects of its size and age. Sample course material is available at www.teachbsd.com (see description following the index).

Exercises are provided at the end of each chapter. The exercises are graded into three categories indicated by zero, one, or two asterisks. The answers to exercises that carry no asterisks can be found in the text. Exercises with a single asterisk require a step of reasoning, critical thinking, or intuition beyond a concept presented in the text. Exercises with two asterisks present major design projects or open research questions.

Organization

This text discusses both philosophical and design issues, as well as details of the system's actual implementation. Often, the discussion starts at the system-call level and descends into the kernel. Tables and figures are used to clarify data structures and control flow. Pseudocode similar to the C language displays algorithms. A bold font identifies program names and filesystem pathnames. A bold and italic font introduces glossary terms. An italic font identifies the names of system calls, variables, routines, and structure names. Routine names (other than system calls) are further identified by the name followed by parentheses (e.g., *malloc*() is the name of a routine, whereas *argv* is the name of a variable).

The book is divided into five parts, organized as follows:

• **Part I, Overview** Three introductory chapters provide the context for the complete operating system and for the rest of the book. Chapter 1, History and Goals, sketches the historical development of the system, emphasizing the

system's research orientation. Chapter 2, Design Overview of FreeBSD, describes the services offered by the system and outlines the internal organization of the kernel. It also discusses the design decisions that were made as the system was developed. Sections 2.3 through 2.15 in Chapter 2 give an overview of their corresponding chapters. Chapter 3, Kernel Services, explains how system calls are performed and describes in detail several of the basic services of the kernel.

- **Part II, Processes** The first chapter in this part—Chapter 4, Process Management—lays the foundation for later chapters by describing the structure of a process, the algorithms used for scheduling the execution of the threads that make up a process, and the synchronization mechanisms used by the system to ensure consistent access to kernel-resident data structures. Chapter 5, Security, explains the security framework used throughout the kernel. It also details the security facilities that are available to control process access to the resources on the system and to each other. In Chapter 6, Memory Management, the virtual-memory-management system is discussed in detail.
- **Part III, I/O System** First, Chapter 7, I/O System Overview, explains the system interface to I/O and describes the structure of the facilities that support this interface. Following this introduction are four chapters that give the details of the main parts of the I/O system. Chapter 8, Devices, gives a description of the I/O architecture of the Intel and AMD systems, and describes how the I/O subsystem is managed and how the kernel initially maps out and later manages the arrival and departure of connected devices. Chapter 9, The Fast Filesystem, details the data structures and algorithms that implement the original local filesystem as seen by application programs, as well as how local filesystems are interfaced with the device interface described in Chapter 8. Chapter 10, The Zettabyte Filesystem, describes the filesystem most recently added to FreeBSD from the OpenSolaris operating system. Chapter 11, The Network Filesystem, explains the latest version 4.2 network filesystem from both the server and client perspectives.
- Part IV, Interprocess Communication Chapter 12, Interprocess Communication, describes the mechanism for providing communication between related or unrelated processes. Chapters 13 and 14, Network-Layer Protocols and Transport-Layer Protocols, are closely related because the facilities explained in the former are used by the protocols, such as the UDP, TCP, and SCTP, explained in the latter.
- Part V, System Operation Chapter 15, System Startup and Shutdown, explains system initialization at the process level from kernel initialization to user login.

The book is intended to be read in the order that the chapters are presented, but the parts other than Part I are independent of one another and can be read separately. Chapter 15 should be read after all the others, but knowledgeable readers may find it useful independently.

At the end of the book are a glossary with brief definitions of major terms and an index. Each chapter contains a Reference section with citations of related material.

Getting BSD

All the BSD distributions are available either for downloading from the net or on removable media such as CD-ROM or DVD. Information on obtaining source and binaries for FreeBSD can be obtained from http://www.FreeBSD.org. The NetBSD distribution is compiled and ready to run on most workstation architectures. For more information, contact the NetBSD Project at http://www.NetBSD.org/. The OpenBSD distribution is compiled and ready to run on a wide variety of workstation architectures and has been extensively vetted for security and reliability. For more information, visit the OpenBSD project's Web site at http://www.OpenBSD.org/.

You diehards that read to the end of the preface are rewarded by nding out that you can get a 32-hour introductory video course based on this book, a 40-hour advanced video course based on the FreeBSD 5 source code, a 2.5-hour video lecture on the history of BSD, and a 4-CD set containing all the releases and the source-control history of BSD from Berkeley. These items are described in the advertisements that follow the index.

Acknowledgments

We extend special thanks to Matt Ahrens (Delphix) who provided invaluable insight on the workings of the ZFS lesystem including countless e-mails answer - ing our questions about how it works and why speci c design decisions were made.

We also thank the following people who provided extensive review of areas of the kernel in which they have deep knowledge: John Baldwin (The FreeBSD Project) on locking, scheduling, and virtual memory; Alan Cox (Rice University) on virtual memory; Jeffrey Roberson (EMC) on the ULE scheduler; and Randall Stewart (Adara Networks) on the SCTP implementation.

We thank the following people, all of whom read and commented on early drafts of various chapters of the book: Eric Allman (University of California, Berkeley); Jonathan Anderson (Memorial University of Newfoundland); David Chisnall (University of Cambridge); Paul Dagnelie (Delphix); Brooks Davis (SRI International); Pawe Jakub Dawidek (Wheel Systems); Peter Grehan (The FreeBSD Project); Scott Long (Net ix); Jak e Luck; Rick Macklem (The FreeBSD Project); Ilias Marinos (University of Cambridge); Roger Pau Monné (Citrex); Mark Robert Vaughan Murray; Edward Tomasz Napiera a (The FreeBSD Project); Peter G. Neumann (SRI International); Rui Paulo; Luigi Rizzo (Universitá di Pisa, Italy); Margo Seltzer (Harvard University); Keith Sklower (University of California, Berkeley); Lawrence Stewart (Swinburne University of Technology); Michael Tuexen (Muenster University of Applied Sciences); Bryan Venteicher (NetApp); Erez Zadok (Stony Brook University); and Bjoern A. Zeeb (The FreeBSD Project).

We are grateful to our now-retired editor of 25 years, Peter Gordon, who had faith in our ability to get the book written despite several years of delays on our part. We are equally grateful to our new editor, Debra Williams, who saw this project to completion and who accelerated the production when we nally had a completed manuscript. We thank all the professional people at Addison-Wesley and Pearson Education who helped us bring the book to completion: managing editor John Fuller; production editor Mary Kesel Wilson; cover designer Chuti Prasertsith; copy editor Deborah Thompson; and proofreader Melissa Panagos. Finally we acknowledge the contributions of Jaap Akkerhuis, who designed the troff macros for the BSD books.

This book was produced using James Clark's implementations of **pic**, **tbl**, **eqn**, and **groff**. The index was generated by **awk** scripts derived from indexing programs written by Jon Bentley and Brian Kernighan [Bentley & Kernighan, 1986]. Most of the art was created with **xfig**. Figure placement and widow elimination were handled by the **groff** macros, but orphan elimination and production of even page bottoms had to be done by hand.

We encourage readers to send us suggested improvements or comments about typographical or other errors found in the book; please send electronic mail to **FreeBSDbook-bugs@McKusick.COM**.

References

Bentley & Kernighan, 1986.

J. Bentley & B. Kernighan, "Tools for Printing Indexes," Computing Science Technical Report 128, AT&T Bell Laboratories, Murray Hill, NJ, October 1986.

Comer, 2000.

D. Comer, *Internetworking with TCP/IP Volume 1, 4th ed.*, Prentice-Hall, Upper Saddle River, NJ, 2000.

FreeBSD Mall, 2004.

FreeBSD Mall, *The FreeBSD Handbook*, available from http://www.freebsdmall.com, March 2004.

FreeBSD.org, 2014.

FreeBSD.org, *The Online FreeBSD Handbook*, available from http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook, March 2014.

Kernighan & Ritchie, 1989.

B. W. Kernighan & D. M. Ritchie, *The C Programming Language*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ, 1989.

Lavigne, 2010.

D. Lavigne, *The Definitive Guide to PC-BSD*, Apress / Springer-Verlag, March 2010.

Organick, 1975.

E. I. Organick, *The Multics System: An Examination of Its Structure*, MIT Press, Cambridge, MA, 1975.

PC-BSD.org, 2014.

PC-BSD.org, *The PC-BSD Users Handbook*, available from http://wiki.pcbsd.org, June 2014.

Ritchie & Thompson, 1978.

D. M. Ritchie & K. Thompson, "The UNIX Time-Sharing System," *Bell System Technical Journal*, vol. 57, no. 6, Part 2, pp. 78–90, July–August

1978. The original version [Comm. ACM vol. 7, no. 7, pp. 365-375 (July

1974)] described the 6th edition; this citation describes the 7th edition. Silberschatz et al., 2012.

A. Silberschatz, P. Galvin, & G. Gagne, *Operating System Concepts*, 9th ed., John Wiley and Sons, Hoboken, NJ, 2012.

Stallings, 2000.

R. Stallings, *Data and Computer Communications*, 6th ed., Prentice Hall, Hoboken, NJ, 2000.

Stevens, 1992.

W. Stevens, Advanced Programming in the UNIX Environment, Addison-Wesley, Reading, MA, 1992.

Tanenbaum, 2010.

A. S. Tanenbaum, *Computer Networks*, 5th ed., Prentice-Hall, Englewood Cliffs, NJ, 2010.

About the Authors



left to right Marshall Kirk McKusick, Robert N.M. Watson, and George V. Neville-Neil

Marshall Kirk McKusick writes books and articles, consults, and teaches classes on UNIX- and BSD-related subjects. While at the University of California at Berkeley, he implemented the 4.2BSD fast filesystem and was the Research Computer Scientist at the Berkeley Computer Systems Research Group (CSRG), overseeing the development and release of 4.3BSD and 4.4BSD. His particular areas of interest are the virtual-memory system and the filesystem. He earned his undergraduate degree in electrical engineering from Cornell University and did his graduate work at the University of California at Berkeley, where he received master's degrees in computer science and business administration, and a doctoral degree in computer science. He has twice been president of the board of the Usenix Association, is currently a member of the FreeBSD Foundation Board of Directors, a member of the editorial board of ACM's Queue magazine, a senior member of the IEEE, and a member of the Usenix Association, ACM, and AAAS. In his spare time, he enjoys swimming, scuba diving, and wine collecting. The wine is stored in a specially constructed wine cellar (accessible from the Web at http://www.McKusick.com/cgi-bin/readhouse) in the basement of the house that he shares with Eric Allman, his partner of 35-and-some-odd years and husband since 2013.

About the Authors

George V. Neville-Neil hacks, writes, teaches, and consults in the areas of Security, Networking, and Operating Systems. Other areas of interest include embedded and real-time systems, network time protocols, and code spelunking. In 2007, he helped start the AsiaBSDCon series of conferences in Tokyo, Japan, and has served on the program committee every year since then. He is a member of the FreeBSD Foundation Board of Directors, and was a member of the FreeBSD Core Team for 4 years. Contributing broadly to open source, he is the lead developer on the Precision Time Protocol project (http://ptpd.sf.net) and the developer of the Packet Construction Set (http://pcs.sf.net). Since 2004, he has written a monthly column, "Kode Vicious," that appears both in ACM's Queue and Communications of the ACM. He serves on the editorial board of ACM's Queue magazine, is vice-chair of ACM's Practitioner Board, and is a member of the Usenix Association, ACM, IEEE, and AAAS. He earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts. He is an avid bicyclist, hiker, and traveler who has lived in Amsterdam, The Netherlands, and Tokyo, Japan. He is currently based in Brooklyn, New York, where he lives with his husband, Kaz Senju.

Robert N.M. Watson is a University Lecturer in Systems, Security, and Architecture in the Security Research Group at the University of Cambridge Computer Laboratory. He supervises doctoral students and postdoctoral researchers in cross-layer research projects spanning computer architecture, compilers, program analysis, program transformation, operating systems, networking, and security. Dr. Watson is a member of the FreeBSD Foundation Board of Directors, was a member of the FreeBSD Core Team for 10 years, and has been a FreeBSD committer for 15 years. His open-source contributions include work on FreeBSD networking, security, and multiprocessing. Having grown up in Washington, D. C., he earned his undergraduate degree in Logic and Computation, with a double major in Computer Science, at Carnegie Mellon University in Pittsburgh, Pennsylvania, and then worked at a series of industrial research labs investigating computer security. He earned his doctoral degree at the University of Cambridge, where his graduate research was in extensible operatingsystem access control. Dr. Watson and his wife Dr. Leigh Denault have lived in Cambridge, England, for 10 years.

This page intentionally left blank

Process Management

4.1 Introduction to Process Management

A *process* is a program in execution. A process has an address space containing a mapping of its program's object code and global variables. It also has a set of kernel resources that it can name and on which it can operate using system calls. These resources include its credentials, signal state, and its descriptor array that gives it access to files, pipes, sockets, and devices. Each process has at least one and possibly many threads that execute its code. Every thread represents a virtual processor with a full context worth of register state and its own stack mapped into the address space. Every thread running in the process has a corresponding kernel thread, with its own kernel stack that represents the user thread when it is executing in the kernel as a result of a system call, page fault, or signal delivery.

A process must have system resources, such as memory and the underlying CPU. The kernel supports the illusion of concurrent execution of multiple processes by scheduling system resources among the set of processes that are ready to execute. On a multiprocessor, multiple threads of the same or different processes may execute concurrently. This chapter describes the composition of a process, the method that the system uses to switch between the process's threads, and the scheduling policy that it uses to promote sharing of the CPU. It also introduces process creation and termination, and details the signal and process-debugging facilities.

Two months after the developers began the first implementation of the UNIX operating system, there were two processes: one for each of the terminals of the PDP-7. At age 10 months, and still on the PDP-7, UNIX had many processes, the *fork* operation, and something like the *wait* system call. A process executed a new program by reading in a new program on top of itself. The first PDP-11 system (First Edition UNIX) saw the introduction of *exec*. All these systems allowed only one process in memory at a time. When a PDP-11 with memory management (a

KS-11) was obtained, the system was changed to permit several processes to remain in memory simultaneously, to reduce swapping. But this change did not apply to multiprogramming because disk I/O was synchronous. This state of affairs persisted into 1972 and the first PDP-11/45 system. True multiprogramming was finally introduced when the system was rewritten in C. Disk I/O for one process could then proceed while another process ran. The basic structure of process management in UNIX has not changed since that time [Ritchie, 1988].

The threads of a process operate in either *user mode* or *kernel mode*. In user mode, a thread executes application code with the machine in a nonprivileged protection mode. When a thread requests services from the operating system with a system call, it switches into the machine's privileged protection mode via a protected mechanism and then operates in kernel mode.

The resources used by a thread are split into two parts. The resources needed for execution in user mode are defined by the CPU architecture and typically include the CPU's general-purpose registers, the program counter, the processorstatus register, and the stack-related registers, as well as the contents of the memory segments that constitute FreeBSD's notion of a program (the text, data, shared library, and stack segments).

Kernel-mode resources include those required by the underlying hardware such as registers, program counter, and the stack pointer. These resources also include the state required for the FreeBSD kernel to provide system services for a thread. This *kernel state* includes parameters to the current system call, the current process's user identity, scheduling information, and so on. As described in Section 3.1, the kernel state for each process is divided into several separate data structures, with two primary structures: the *process structure* and the *thread structure*.

The process structure contains information that must always remain resident in main memory, along with references to other structures that remain resident, whereas the thread structure tracks information that needs to be resident only when the process is executing such as its kernel run-time stack. Process and thread structures are allocated dynamically as part of process creation and are freed when the process is destroyed as it exits.

Multiprogramming

FreeBSD supports transparent multiprogramming: the illusion of concurrent execution of multiple processes or programs. It does so by *context switching*—that is, by switching between the execution context of the threads within the same or different processes. A mechanism is also provided for *scheduling* the execution of threads—that is, for deciding which one to execute next. Facilities are provided for ensuring consistent access to data structures that are shared among processes.

Context switching is a hardware-dependent operation whose implementation is influenced by the underlying hardware facilities. Some architectures provide machine instructions that save and restore the hardware-execution context of a thread or an entire process including its virtual-address space. On others, the software must collect the hardware state from various registers and save it, then load those registers with the new hardware state. All architectures must save and restore the software state used by the kernel.

Context switching is done frequently, so increasing the speed of a context switch noticeably decreases time spent in the kernel and provides more time for execution of user applications. Since most of the work of a context switch is expended in saving and restoring the operating context of a thread or process, reducing the amount of the information required for that context is an effective way to produce faster context switches.

Scheduling

Fair scheduling of threads and processes is an involved task that is dependent on the types of executable programs and on the goals of the scheduling policy. Programs are characterized according to the amount of computation and the amount of I/O that they do. Scheduling policies typically attempt to balance resource utilization against the time that it takes for a program to complete. In FreeBSD's default scheduler, which we shall refer to as the timeshare scheduler, a process's priority is periodically recalculated based on various parameters, such as the amount of CPU time it has used, the amount of memory resources it holds or requires for execution, etc. Some tasks require more precise control over process execution called real-time scheduling. Real-time scheduling must ensure that threads finish computing their results by a specified deadline or in a particular order. The FreeBSD kernel implements real-time scheduling using a separate queue from the queue used for regular timeshared processes. A process with a real-time priority is not subject to priority degradation and will only be preempted by another thread of equal or higher real-time priority. The FreeBSD kernel also implements a queue of threads running at idle priority. A thread with an idle priority will run only when no other thread in either the real-time or timeshare-scheduled queues is runnable and then only if its idle priority is equal to or greater than all other runnable idle-priority threads.

The FreeBSD timeshare scheduler uses a priority-based scheduling policy that is biased to favor *interactive programs*, such as text editors, over long-running batch-type jobs. Interactive programs tend to exhibit short bursts of computation followed by periods of inactivity or I/O. The scheduling policy initially assigns a high execution priority to each thread and allows that thread to execute for a fixed *time slice*. Threads that execute for the duration of their slice have their priority lowered, whereas threads that give up the CPU (usually because they do I/O) are allowed to remain at their priority. Threads that are inactive have their priority raised. Jobs that use large amounts of CPU time sink rapidly to a low priority, whereas interactive jobs that are mostly inactive remain at a high priority so that, when they are ready to run, they will preempt the long-running lower-priority jobs. An interactive job, such as a text editor searching for a string, may become compute-bound briefly and thus get a lower priority, but it will return to a high priority when it is inactive again while the user thinks about the result.

Some tasks, such as the compilation of a large application, may be done in many small steps in which each component is compiled in a separate process. No

individual step runs long enough to have its priority degraded, so the compilation as a whole impacts the interactive programs. To detect and avoid this problem, the scheduling priority of a child process is propagated back to its parent. When a new child process is started, it begins running with its parent's current priority. As the program that coordinates the compilation (typically **make**) starts many compilation steps, its priority is dropped because of the CPU-intensive behavior of its children. Later compilation steps started by **make** begin running and stay at a lower priority, which allows higher-priority interactive programs to run in preference to them as desired.

The system also needs a scheduling policy to deal with problems that arise from not having enough main memory to hold the execution contexts of all processes that want to execute. The major goal of this scheduling policy is to minimize *thrashing*—a phenomenon that occurs when memory is in such short supply that more time is spent in the system handling page faults and scheduling processes than in user mode executing application code.

The system must both detect and eliminate thrashing. It detects thrashing by observing the amount of free memory. When the system has little free memory and a high rate of new memory requests, it considers itself to be thrashing. The system reduces thrashing by marking the least recently run process as not being allowed to run, allowing the pageout daemon to push all the pages associated with the process to backing store. On most architectures, the kernel also can push to backing store the kernel stacks of all the threads of the marked process. The effect of these actions is to cause the process and all its threads to be swapped out (see Section 6.12). The memory freed by blocking the process can then be distributed to the remaining processes, which usually can then proceed. If the thrashing continues, additional processes are selected to be blocked from running until enough memory becomes available for the remaining processes to run effectively. Eventually, enough processes complete and free their memory that blocked processes can resume execution. However, even if there is not enough memory, the blocked processes are allowed to resume execution after about 20 seconds. Usually, the thrashing condition will return, requiring that some other process be selected for being blocked (or that an administrative action be taken to reduce the load).

4.2 Process State

Every process in the system is assigned a unique identifier termed the *process identifier* (*PID*). PIDs are the common mechanism used by applications and by the kernel to reference processes. PIDs are used by applications when the latter send a signal to a process and when receiving the exit status from a deceased process. Two PIDs are of special importance to each process: the PID of the process itself and the PID of the process's parent process.

The layout of process state is shown in Figure 4.1. The goal is to support multiple threads that share an address space and other resources. A *thread* is the unit of execution of a process; it requires an address space and other resources, but it can share many of those resources with other threads. Threads sharing an


address space and other resources are scheduled independently and in FreeBSD can all execute system calls simultaneously. The process state in FreeBSD is designed to support threads that can select the set of resources to be shared, known as variable-weight processes [Aral et al., 1989].

Each of the components of process state is placed into separate substructures for each type of state information. The process structure references all the substructures directly or indirectly. The thread structure contains just the information needed to run in the kernel: information about scheduling, a stack to use when running in the kernel, a *thread state block* (*TSB*), and other machine-dependent state. The TSB is defined by the machine architecture; it includes the general-purpose registers, stack pointers, program counter, processor-status word, and memory-management registers.

The first threading models that were deployed in systems such as FreeBSD 5 and Solaris used an N:M threading model in which many user level threads (N) were supported by a smaller number of threads (M) that could run in the kernel [Simpleton, 2008]. The N:M threading model was light-weight but incurred extra overhead when a user-level thread needed to enter the kernel. The model assumed that application developers would write server applications in which potentially thousands of clients would each use a thread, most of which would be idle waiting for an I/O request.

While many of the early applications using threads, such as file servers, worked well with the N:M threading model, later applications tended to use pools of dozens to hundreds of worker threads, most of which would regularly enter the kernel. The application writers took this approach because they wanted to run on a wide range of platforms and key platforms like Windows and Linux could not support tens of thousands of threads. For better efficiency with these applications, the N:M threading model evolved over time to a 1:1 threading model in which every user thread is backed by a kernel thread.

Like most other operating systems, FreeBSD has settled on using the POSIX threading API often referred to as Pthreads. The Pthreads model includes a rich set of primitives including the creation, scheduling, coordination, signalling, rendezvous, and destruction of threads within a process. In addition, it provides shared and exclusive locks, semaphores, and condition variables that can be used to reliably interlock access to data structures being simultaneously accessed by multiple threads.

In their lightest-weight form, FreeBSD threads share all the process resources including the PID. When additional parallel computation is needed, a new thread is created using the *pthread_create()* library call. The pthread library must keep track of the user-level stacks being used by each of the threads, since the entire address space is shared including the area normally used for the stack. Since the threads all share a single process structure, they have only a single PID and thus show up as a single entry in the **ps** listing. There is an option to **ps** that requests it to list a separate entry for each thread within a process.

Many applications do not wish to share all of a process's resources. The *rfork* system call creates a new process entry that shares a selected set of resources from its parent. Typically, the signal actions, statistics, and the stack and data parts of the address space are not shared. Unlike the lightweight thread created by *pthread_create()*, the *rfork* system call associates a PID with each thread that shows up in a **ps** listing and that can be manipulated in the same way as any other process in the system. Processes created by *fork*, *vfork*, or *rfork* initially have just a single thread structure associated with them. A variant of the *rfork* system call is used to emulate the Linux *clone()* functionality.

The Process Structure

In addition to the references to the substructures, the process entry shown in Figure 4.1 contains the following categories of information:

- Process identification: the PID and the parent PID
- Signal state: signals pending delivery and summary of signal actions
- Tracing: process tracing information
- Timers: real-time timer and CPU-utilization counters

The process substructures shown in Figure 4.1 have the following categories of information:

- Process-group identification: the process group and the session to which the process belongs
- User *credential*s: the real, effective, and saved user and group identifiers; credentials are described more fully in Chapter 5
- Memory management: the structure that describes the allocation of virtual address space used by the process; the virtual-address space and its related structures are described more fully in Chapter 6
- File descriptors: an array of pointers to file entries indexed by the process's open file descriptors; also, the open file flags and current directory
- System call vector: the mapping of system call numbers to actions; in addition to current and deprecated native FreeBSD executable formats, the kernel can run binaries compiled for several other UNIX variants such as Linux and System V Release 4 by providing alternative system call vectors when such environments are requested
- Resource accounting: the *rlimit* structures that describe the utilization of the many resources provided by the system (see Section 3.7)
- Statistics: statistics collected while the process is running that are reported when it exits and are written to the accounting file; also includes process timers and profiling information if the latter is being collected
- Signal actions: the action to take when a signal is posted to a process
- Thread structure: the contents of the thread structure (described at the end of this section)

The state element of the process structure holds the current value of the process state. The possible state values are shown in Table 4.1. When a process is first

Table 4.1 Process states.

State	Description
NEW	undergoing process creation
NORMAL	thread(s) will be RUNNABLE, SLEEPING, or STOPPED
ZOMBIE	undergoing process termination

created with a *fork* system call, it is initially marked as NEW. The state is changed to NORMAL when enough resources are allocated to the process for the latter to begin execution. From that point onward, a process's state will be NORMAL until the process terminates. Its thread(s) will fluctuate among RUNNABLE—that is, preparing to be or actually executing; SLEEPING—that is, waiting for an event; and STOPPED—that is, stopped by a signal or the parent process. A deceased process is marked as ZOMBIE until it has freed its resources and communicated its termination status to its parent process.

The system organizes process structures into two lists. Process entries are on the *zombproc* list if the process is in the ZOMBIE state; otherwise, they are on the *allproc* list. The two queues share the same linkage pointers in the process structure, since the lists are mutually exclusive. Segregating the dead processes from the live ones reduces the time spent both by the *wait* system call, which must scan the zombies for potential candidates to return, and by the scheduler and other functions that must scan all the potentially runnable processes.

Most threads, except the currently executing thread (or threads if the system is running on a multiprocessor), are also in one of three queues: a *run queue*, a *sleep queue*, or a *turnstile queue*. Threads that are in a runnable state are placed on a run queue, whereas threads that are blocked while awaiting an event are located on either a turnstile queue or a sleep queue. Stopped threads awaiting an event are located on a turnstile queue, a sleep queue, or they are on no queue. The run queues are organized according to thread-scheduling priority and are described in Section 4.4. The sleep and turnstile queues are organized in a data structure that is hashed by an event identifier. This organization optimizes finding the sleeping threads that need to be awakened when a wakeup occurs for an event. The sleep and turnstile queues are described in Section 4.3.

The p_pptr pointer and related lists ($p_children$ and $p_sibling$) are used in locating related processes, as shown in Figure 4.2. When a process spawns a child process, the child process is added to its parent's $p_children$ list. The child process also keeps a backward link to its parent in its p_pptr pointer. If a process has more than one child process active at a time, the children are linked together through their $p_sibling$ list entries. In Figure 4.2, process B is a direct descendant of process A, whereas processes C, D, and E are descendants of process B and are

Figure 4.2 Process-group hierarchy.



Range	Class	Thread type
0 - 47	ITHD	bottom-half kernel (interrupt)
48 – 79	REALTIME	real-time user
80 - 119	KERN	top-half kernel
120 - 223	3 TIMESHARE	time-sharing user
224 - 255	5 IDLE	idle user

Table 4.2 Thread-scheduling classes.

siblings of one another. Process B typically would be a shell that started a pipeline (see Sections 2.4 and 4.8) including processes C, D, and E. Process A probably would be the system-initialization process **init** (see Sections 3.1 and 15.4).

CPU time is made available to threads according to their *scheduling class* and *scheduling priority*. As shown in Table 4.2, the FreeBSD kernel has two kernel and three user scheduling classes. The kernel will always run the thread in the highest-priority class. Any kernel-interrupt threads will run in preference to any-thing else followed by any runnable real-time threads. Any top-half-kernel threads are run in preference to runnable threads in the share and idle classes. Runnable timeshare threads are run in preference to runnable threads in the idle class. The priorities of threads in the real-time and idle classes are set by the applications using the *rtprio* system call and are never adjusted by the kernel. The bottom-half interrupt priorities are set based on predefined priorities for each kernel subsystem and never change.

The priorities of threads running in the timeshare class are adjusted by the kernel based on resource usage and recent CPU utilization. A thread has two scheduling priorities: one for scheduling user-mode execution and one for scheduling kernel-mode execution. The td_user_pri field associated with the thread structure contains the user-mode scheduling priority, whereas the $td_priority$ field holds the current scheduling priority. The current priority may be different from the user-mode priority when the thread is executing in the top half of the kernel. Priorities range between 0 and 255, with a lower value interpreted as a higher priority (see Table 4.2). User-mode priorities range from 120 to 255; priorities less than 120 are used only by real-time threads or when a thread is asleep—that is, awaiting an event in the kernel—and immediately after such a thread is awakened. Threads asleep in the kernel are given a higher priority because they typically hold shared kernel resources when they awaken. The system wants to run them as quickly as possible once they get a resource so that they can use the resource and return it before another thread requests it and gets blocked waiting for it.

When a thread goes to sleep in the kernel, it must specify whether it should be awakened and marked runnable if a signal is posted to it. In FreeBSD, a kernel thread will be awakened by a signal only if it sets the PCATCH flag when it sleeps. The *msleep()* interface also handles sleeps limited to a maximum time duration and the processing of restartable system calls. The *msleep()* interface includes a reference to a string describing the event that the thread awaits; this string is externally visible—for example, in **ps**. The decision of whether to use an interruptible sleep depends on how long the thread may be blocked. Because it is complex to handle signals in the midst of doing some other operation, many sleep requests are not interruptible; that is, a thread will not be scheduled to run until the event for which it is waiting occurs. For example, a thread waiting for disk I/O will sleep with signals blocked.

For quickly occurring events, delaying to handle a signal until after they complete is imperceptible. However, requests that may cause a thread to sleep for a long period, such as waiting for terminal or network input, must be prepared to have its sleep interrupted so that the posting of signals is not delayed indefinitely. Threads that sleep interruptibly may abort their system call because of a signal arriving before the event for which they are waiting has occurred. To avoid holding a kernel resource permanently, these threads must check why they have been awakened. If they were awakened because of a signal, they must release any resources that they hold. They must then return the error passed back to them by sleep(), which will be EINTR if the system call is to be aborted after the signal or ERESTART if it is to be restarted. Occasionally, an event that is supposed to occur quickly, such as a disk I/O, will get held up because of a hardware failure. Because the thread is sleeping in the kernel with signals blocked, it will be impervious to any attempts to send it a signal, even a signal that should cause it to exit unconditionally. The only solution to this problem is to change *sleep()*s on hardware events that may hang to be interruptible.

In the remainder of this book, we shall always use sleep() when referring to the routine that puts a thread to sleep, even when one of the $mtx_sleep()$, $sx_sleep()$, $rw_sleep()$, or $t_sleep()$ interfaces is the one that is being used.

The Thread Structure

The thread structure shown in Figure 4.1 contains the following categories of information:

- Scheduling: the thread priority, user-mode scheduling priority, recent CPU utilization, and amount of time spent sleeping; the run state of a thread (runnable, sleeping); additional status flags; if the thread is sleeping, the *wait channel*, the identity of the event for which the thread is waiting (see Section 4.3), and a pointer to a string describing the event
- TSB: the user- and kernel-mode execution states
- Kernel stack: the per-thread execution stack for the kernel
- Machine state: the machine-dependent thread information

Historically, the kernel stack was mapped to a fixed location in the virtual address space. The reason for using a fixed mapping is that when a parent forks, its run-

time stack is copied for its child. If the kernel stack is mapped to a fixed address, the child's kernel stack is mapped to the same addresses as its parent kernel stack. Thus, all its internal references, such as frame pointers and stack-variable references, work as expected.

On modern architectures with virtual address caches, mapping the kernel stack to a fixed address is slow and inconvenient. FreeBSD removes this constraint by eliminating all but the top call frame from the child's stack after copying it from its parent so that it returns directly to user mode, thus avoiding stack copying and relocation problems.

Every thread that might potentially run must have its stack resident in memory because one task of its stack is to handle page faults. If it were not resident, it would page fault when the thread tried to run, and there would be no kernel stack available to service the page fault. Since a system may have many thousands of threads, the kernel stacks must be kept small to avoid wasting too much physical memory. In FreeBSD on the Intel architecture, the kernel stack is limited to two pages of memory. Implementors must be careful when writing code that executes in the kernel to avoid using large local variables and deeply nested subroutine calls to avoid overflowing the run-time stack. As a safety precaution, some architectures leave an invalid page between the area for the run-time stack and the data structures that follow it. Thus, overflowing the kernel stack will cause a kernelaccess fault instead of disastrously overwriting other data structures. It would be possible to simply kill the process that caused the fault and continue running. However, the cleanup would be difficult because the thread may be holding locks or be in the middle of modifying some data structure that would be left in an inconsistent or invalid state. So the FreeBSD kernel panics on a kernel-access fault because such a fault shows a fundamental design error in the kernel. By panicking and creating a crash dump, the error can usually be pinpointed and corrected.

4.3 Context Switching

The kernel switches among threads in an effort to share the CPU effectively; this activity is called *context switching*. When a thread executes for the duration of its time slice or when it blocks because it requires a resource that is currently unavailable, the kernel finds another thread to run and context switches to it. The system can also interrupt the currently executing thread to run a thread triggered by an asynchronous event, such as a device interrupt. Although both scenarios involve switching the execution context of the CPU, switching between threads occurs *synchronously* with respect to the currently executing thread, whereas servicing interrupts occurs *asynchronously* with respect to the current thread. In addition, interprocess context switches are classified as voluntary or involuntary. A voluntary context switch occurs when a thread blocks because it requires a resource that is unavailable. An involuntary context switch takes place when a thread executes for the duration of its time slice or when the system identifies a higher-priority thread to run.

Each type of context switching is done through a different interface. Voluntary context switching is initiated with a call to the sleep() routine, whereas an involuntary context switch is forced by direct invocation of the low-level context-switching mechanism embodied in the $mi_switch()$ and setrunnable() routines. Asynchronous event handling is triggered by the underlying hardware and is effectively transparent to the system.

Thread State

Context switching between threads requires that both the kernel- and user-mode context be changed. To simplify this change, the system ensures that all of a thread's user-mode state is located in the thread structure while most kernel state is kept elsewhere. The following conventions apply to this localization:

- Kernel-mode hardware-execution state: Context switching can take place in only kernel mode. The kernel's hardware-execution state is defined by the contents of the TSB that is located in the thread structure.
- User-mode hardware-execution state: When execution is in kernel mode, the user-mode state of a thread (such as copies of the program counter, stack pointer, and general registers) always resides on the kernel's execution stack that is located in the thread structure. The kernel ensures this location of user-mode state by requiring that the system-call and trap handlers save the contents of the user-mode execution context each time that the kernel is entered (see Section 3.1).
- The process structure: The process structure always remains resident in memory.
- Memory resources: Memory resources of a process are effectively described by the contents of the memory-management registers located in the TSB and by the values present in the process and thread structures. As long as the process remains in memory, these values will remain valid and context switches can be done without the associated page tables being saved and restored. However, these values need to be recalculated when the process returns to main memory after being swapped to secondary storage.

Low-Level Context Switching

The localization of a process's context in that process's thread structure permits the kernel to perform context switching simply by changing the notion of the current thread structure and (if necessary) process structure, and restoring the context described by the TSB within the thread structure (including the mapping of the virtual address space). Whenever a context switch is required, a call to the $mi_switch()$ routine causes the highest-priority thread to run. The $mi_switch()$ routine first selects the appropriate thread from the scheduling queues, and then resumes the selected thread by loading its context from its TSB.

Voluntary Context Switching

A voluntary context switch occurs whenever a thread must await the availability of a resource or the arrival of an event. Voluntary context switches happen frequently in normal system operation. In FreeBSD, voluntary context switches are initiated through a request to obtain a lock that is already held by another thread or by a call to the *sleep()* routine. When a thread no longer needs the CPU, it is suspended, awaiting the resource described by a *wait channel*, and is given a scheduling priority that should be assigned to the thread when that thread is awakened. This priority does not affect the user-level scheduling priority.

When blocking on a lock, the wait channel is usually the address of the lock. When blocking for a resource or an event, the wait channel is typically the address of some data structure that identifies the resource or event for which the thread is waiting. For example, the address of a disk buffer is used while the thread is waiting for the buffer to be filled. When the buffer is filled, threads sleeping on that wait channel will be awakened. In addition to the resource addresses that are used as wait channels, there are some addresses that are used for special purposes:

- When a parent process does a *wait* system call to collect the termination status of its children, it must wait for one of those children to exit. Since it cannot know which of its children will exit first, and since it can sleep on only a single wait channel, there is a quandary about how to wait for the next of multiple events. The solution is to have the parent sleep on its own process structure. When a child exits, it awakens its parent's process-structure address rather than its own. Thus, the parent doing the *wait* will awaken independently of which child process is the first to exit. Once running, it must scan its list of children to determine which one exited.
- When a thread does a *sigsuspend* system call, it does not want to run until it receives a signal. Thus, it needs to do an interruptible sleep on a wait channel that will never be awakened. By convention, the address of the signal-actions structure is given as the wait channel.

A thread may block for a short, medium, or long period of time depending on the reason that it needs to wait. A short wait occurs when a thread needs to wait for access to a lock that protects a data structure. A medium wait occurs while a thread waits for an event that is expected to occur quickly such as waiting for data to be read from a disk. A long wait occurs when a thread is waiting for an event that will happen at an indeterminate time in the future such as input from a user.

Short-term waits arise only from a lock request. Short-term locks include mutexes, read-writer locks, and read-mostly locks. Details on these locks are given later in this section. A requirement of short-term locks is that they may not be held while blocking for an event as is done for medium- and long-term locks. The only reason that a thread holding a short-term lock is not running is that it has been preempted by a higher-priority thread. It is always possible to get a short-

term lock released by running the thread that holds it and any threads that block the thread that holds it.

A short-term lock is managed by a *turnstile* data structure. The *turnstile* tracks the current owner of the lock and the list of threads waiting for access to the lock. Figure 4.3 shows how *turnstiles* are used to track blocked threads. Across the top of the figure is a set of hash headers that allow a quick lookup to find a lock with waiting threads. If a *turnstile* is found, it provides a pointer to the thread



Figure 4.3 Turnstile structures for blocked threads.

that currently owns the lock and lists of the threads that are waiting for exclusive and shared access. The most important use of the *turnstile* is to quickly find the threads that need to be awakened when a lock is released. In Figure 4.3, Lock 18 is owned by thread 1 and has threads 2 and 3 waiting for exclusive access to it. The *turnstile* in this example also shows that thread 1 holds contested Lock 15.

A *turnstile* is needed each time a thread blocks on a contested lock. Because blocking is common, it would be prohibitively slow to allocate and free a *turnstile* every time one is needed. So each thread allocates a *turnstile* when it is created. As a thread may only be blocked on one lock at any point in time, it will never need more than one *turnstile*. *Turnstiles* are allocated by threads rather than being incorporated into each lock structure because there are far more locks in the kernel than there are threads. Allocating one *turnstile* per thread rather than one per lock results in lower memory utilization in the kernel.

When a thread is about to block on a short-term lock, it provides its *turnstile* to be used to track the lock. If it is the first thread to block on the lock, its *turnstile* is used. If it is not the first thread to block, then an earlier thread's *turnstile* will be in use to do the tracking. The additional *turnstiles* that are provided are kept on a free list whose head is the *turnstile* being used to track the lock. When a thread is awakened and is being made runnable, it is given a *turnstile* from the free list (which may not be the same one that it originally provided). When the last thread is awakened, the free list will be empty and the *turnstile* no longer needed, so it can be taken by the awakening thread.

In Figure 4.3, the *turnstile* tracking Lock 18 was provided by thread 2 as it was the first to block. The spare *turnstile* that it references was provided by thread 3. If thread 2 is the first to be awakened, it will get the spare *turnstile* provided by thread 3 and when thread 3 is awakened later, it will be the last to be awakened so will get the no-longer-needed *turnstile* originally provided by thread 2.

A *priority inversion* occurs when a thread trying to acquire a short-term lock finds that the thread holding the lock has a lower priority than its own priority. The owner and list of blocked threads tracked by the *turnstile* allows *priority propagation* of the higher priority from the thread that is about to be blocked to the thread that holds the lock. With the higher priority, the thread holding the lock will run, and if, in turn, it is blocked by a thread with lower priority, it will propagate its new higher priority to that thread. When finished with its access to the protected data structure, the thread with the temporarily raised priority will release the lock. As part of releasing the lock, the propagated priority will be dropped, which usually results in the thread from which the priority was propagated getting to run and now being able to acquire the lock.

Processes blocking on medium-term and long-term locks use *sleepqueue* data structures rather than *turnstiles* to track the blocked threads. The *sleepqueue* data structure is similar to the *turnstile* except that it does not need to track the owner of the lock. The owner need not be tracked because *sleepqueues* do not need to provide priority propagation. Threads blocked on medium- and long-term locks cannot proceed until the event for which they are waiting has occurred. Raising their priority will not allow them to run any sooner.

Sleepqueues have many similarities to *turnstiles* including a hash table to allow quick lookup of contested locks and lists of the threads blocked because they are awaiting shared and exclusive locks. When created, each thread allocates a *sleepqueue* structure. It provides its *sleepqueue* structure when it is about to be put to sleep and is returned a *sleepqueue* structure when it is awakened.

Unlike short-term locks, the medium- and long-term locks can request a time limit so that if the event for which they are waiting has not occurred within the specified period of time, they will be awakened with an error return that indicates that the time limit expired rather than the event occurring. Finally, long-term locks can request that they be interruptible, meaning that they will be awakened if a signal is sent to them before the event for which they are waiting has occurred.

Suspending a thread takes the following steps in its operation:

- 1. Prevents events that might cause thread-state transitions. Historically a global scheduling lock was used, but it was a bottleneck. Now each thread uses a lock tied to its current state to protect its per-thread state. For example, when a thread is on a run queue, the lock for that run queue is used; when the thread is blocked on a *turnstile*, the *turnstile*'s lock is used; when a thread is blocked on a sleep queue, the lock for the wait channels hash chain is used.
- 2. Records the wait channel in the thread structure and hashes the wait-channel value to check for an existing *turnstile* or *sleepqueue* for the wait-channel. If one exists, links the thread to it and saves the *turnstile* or *sleepqueue* structure provided by the thread. Otherwise places the *turnstile* or *sleepqueue* onto the hash chain and links the thread into it.
- 3. For threads being placed on a *turnstile*, if the current thread's priority is higher than the priority of the thread currently holding the lock, propagates the current thread's priority to the thread currently holding the lock. For threads being placed on a *sleepqueue*, sets the thread's priority to the priority that the thread will have when the thread is awakened and sets its SLEEPING flag.
- 4. For threads being placed on a *turnstile*, sort the thread into the list of waiting threads such that the highest priority thread appears first in the list. For threads being placed on a *sleepqueue*, place the thread at the end of the list of threads waiting for that wait-channel.
- 5. Calls *mi_switch()* to request that a new thread be scheduled; the associated mutex is released as part of switching to the other thread.

A sleeping thread is not selected to execute until it is removed from a *turnstile* or *sleepqueue* and is marked runnable. This operation is done either implicitly as part of a lock being released, or explicitly by a call to the *wakeup()* routine to signal that an event has occurred or that a resource is available. When *wakeup()* is invoked, it is given a wait channel that it uses to find the corresponding *sleepqueue* (using a hashed lookup). It awakens all threads sleeping on that wait channel. All threads waiting for the resource are awakened to ensure that none are

inadvertently left sleeping. If only one thread were awakened, it might not request the resource on which it was sleeping. If it does not use and release the resource, any other threads waiting for that resource will be left sleeping forever. A thread that needs an empty disk buffer in which to write data is an example of a thread that may not request the resource on which it was sleeping. Such a thread can use any available buffer. If none is available, it will try to create one by requesting that a dirty buffer be written to disk and then waiting for the I/O to complete. When the I/O finishes, the thread will awaken and will check for an empty buffer. If several are available, it may not use the one that it cleaned, leaving any other threads to sleep forever as they wait for the cleaned buffer.

In instances where a thread will always use a resource when it becomes available, *wakeup_one()* can be used instead of *wakeup()*. The *wakeup_one()* routine wakes up only the first thread that it finds waiting for a resource as it will have been asleep the longest. The assumption is that when the awakened thread is done with the resource, it will issue another *wakeup_one()* to notify the next waiting thread that the resource is available. The succession of *wakeup_one()* calls will continue until all threads waiting for the resource have been awakened and had a chance to use it. Because the threads are ordered from longest to shortest waiting, that is the order in which they will be awakened and gain access to the resource.

When releasing a turnstile lock, all waiting threads are released. Because the threads are ordered from highest to lowest priority, that is the order in which they will be awakened. Usually they will then be scheduled in the order in which they were released. When threads end up being run concurrently, the adaptive spinning (described later in this section) usually ensures that they will not block. And because they are released from highest to lowest priority, the highest priority thread will usually be the first to acquire the lock. There will be no need for, and hence no overhead from, priority propagation. Rather, the lock will be handed down from the highest priority threads through the intermediate priorities to the lowest priority.

To avoid having excessive numbers of threads awakened, kernel programmers try to use locks and wait channels with fine-enough granularity that unrelated uses will not collide on the same resource. For example, they put locks on each buffer in the buffer cache rather than putting a single lock on the buffer cache as a whole.

Resuming a thread takes the following steps in its operation:

- 1. Removes the thread from its *turnstile* or *sleepqueue*. If it is the last thread to be awakened, the *turnstile* or *sleepqueue* is returned to it. If it is not the last thread to be awakened, a *turnstile* or *sleepqueue* from the free list is returned to it.
- 2. Recomputes the user-mode scheduling priority if the thread has been sleeping longer than one second.
- 3. If the thread had been blocked on a *turnstile*, it is placed on the run queue. If the thread had been blocked on a *sleepqueue*, it is placed on the run queue if it is in a SLEEPING state and if its process is not swapped out of main memory.

If the process has been swapped out, the *swapin* process will be awakened to load it back into memory (see Section 6.12); if the thread is in a STOPPED state, it is not put on a run queue until it is explicitly restarted by a user-level process, either by a *ptrace* system call (see Section 4.9) or by a continue signal (see Section 4.7).

If any threads are placed on the run queue and one of them has a scheduling priority higher than that of the currently executing thread, it will also request that the CPU be rescheduled as soon as possible.

Synchronization

The FreeBSD kernel supports both *symmetric multiprocessing* (SMP) and *nonuniform memory access* (NUMA) architectures. An SMP architecture is one in which all the CPUs are connected to a common main memory while a NUMA architecture is one in which the CPUs are connected to a non-uniform memory. With a NUMA architecture, some memory is local to a CPU and is quickly accessible while other memory is slower to access because it is local to another CPU or shared between CPUs. Throughout this book, references to multiprocessors and multiprocessing refer to both SMP and NUMA architectures.

A multiprocessing kernel requires extensive and fine-grained synchronization. The simplist form of synchronization is a critical section. While a thread is running in a critical section, it can neither be migrated to another CPU nor preempted by another thread. A critical section protects per-CPU data structures such as a run queue or CPU-specific memory-allocation data structures. A critical section controls only a single CPU, so it cannot protect systemwide data structures; one of the locking mechanisms described below must be used. While critical sections are useful for only a limited set of data structures, they are beneficial in those cases

Level	Туре	Sleep	Description
Highest	witness	yes	partially ordered sleep locks
	lock manager	yes	drainable shared/exclusive access
	condition variables	yes	event-based thread blocking
	shared-exclusive lock	yes	shared and exclusive access
	read-mostly lock	no	optimized for read access
	reader-writer lock	no	shared and exclusive access
	sleep mutex	no	spin for a while, then sleep
	spin mutex	no	spin lock
Lowest	hardware	no	memory-interlocked compare-and-swap

Table 4.3 Locking hierarchy.

because they have signi cantly lower overhead than locks. A critical section begins by calling *critical_enter()* and continues until calling the function *critical_exit()*.

Table 4.3 shows the hierarchy of locking that is necessary to support multiprocessing. The column labelled Sleep in Table 4.3 shows whether a lock of that type may be held when a thread blocks for a medium- or long-term sleep.

Although it is possible to build locks using single-memory operations [Dekker, 2013], to be practical, the hardware must provide a memory interlocked compare-and-swap instruction. The compare-and-swap instruction must allow two operations to be done on a main-memory location—the reading and comparing to a speci ed compare-v alue of the existing value followed by the writing of a new value if the read value matches the compare-value—without any other processor being able to read or write that memory location between the two memory operations. All the locking primitives in the FreeBSD system are built using the compare-and-swap instruction.

Mutex Synchronization

Mutexes are the primary method of short-term thread synchronization. The major design considerations for mutexes are as follows:

- Acquiring and releasing uncontested mutexes should be as fast as possible.
- Mutexes must have the information and storage space to support priority propagation. In FreeBSD, mutexes use *turnstiles* to manage priority propagation.
- A thread must be able to acquire a mutex recursively if the mutex is initialized to support recursion.

Mutexes are built from the hardware compare-and-swap instruction. A memory location is reserved for the lock. When the lock is free, the value of MTX_UNOWNED is stored in the memory location; when the lock is held, a pointer to the thread owning the lock is stored in the memory location. The compare-andswap instruction tries to acquire the lock. The value in the lock is compared with MTX_UNOWNED; if it matches, it is replaced with the pointer to the thread. The instruction returns the old value; if the old value was MTX_UNOWNED, then the lock was successfully acquired and the thread may proceed. Otherwise, some other thread held the lock so the thread must loop doing the compare-and-swap until the thread holding the lock (and running on a different processor) stores MTX_UNOWNED into the lock to show that it is done with it.

There are currently two avors of mutexes: those that block and those that do not. By default, threads will block when they request a mutex that is already held. Most kernel code uses the default lock type that allows the thread to be suspended from the CPU if it cannot get the lock.

Mutexes that do not sleep are called *spin mutexes*. A spin mutex will not relinquish the CPU when it cannot immediately get the requested lock, but it will

loop, waiting for the mutex to be released by another CPU. Spinning can result in deadlock if a thread interrupted the thread that held a mutex and then tried to acquire the mutex. To protect an interrupt thread from blocking against itself during the period that it is held, a spin mutex runs inside a critical section with interrupts disabled on that CPU. Thus, an interrupt thread can run only on another CPU during the period that the spin mutex is held.

Spin mutexes are specialized locks that are intended to be held for short periods of time. A thread may hold multiple spin mutexes, but it is required to release the mutexes in the opposite order from which they were acquired. A thread may not go to sleep while holding a spin mutex.

On most architectures, both acquiring and releasing an uncontested spin mutex are more expensive than the same operation on a nonspin mutex. Spin mutexes are more expensive than blocking locks because spin mutexes have to disable or defer interrupts while they are held to prevent races with interrupt handling code. As a result, holding spin mutexes can increase interrupt latency. To minimize interrupt latency and reduce locking overhead, FreeBSD uses spin mutexes only in code that does low-level scheduling and context switching.

The time to acquire a lock can vary. Consider the time to wait for a lock needed to search for an item on a list. The thread holding the search lock may have to acquire another lock before it can remove an item it has found from the list. If the needed lock is already held, it will block to wait for it. A different thread that tries to acquire the search lock uses adaptive spinning. Adaptive spinning is implemented by having the thread that wants the lock extract the thread pointer of the owning thread from the lock structure. It then checks to see if the thread is currently executing. If so, it spins until either the lock is released or the thread stops executing. The effect is to spin so long as the current lock holder is executing on another CPU. The reasons for taking this approach are many:

- Locks are usually held for brief periods of time, so if the owner is running, then it will probably release the lock before the current thread completes the process of blocking on the lock.
- If a lock holder is not running, then the current thread has to wait at least one context switch time before it can acquire the lock.
- If the owner is on a run queue, then the current thread should block immediately so it can lend its priority to the lock owner.
- It is cheaper to release an uncontested lock with a single atomic operation than a contested lock. A contested lock has to find the *turnstile*, lock the turnstile chain and *turnstile*, and then awaken all the waiters. So adaptive spinning reduces overhead on both the lock owner and the thread trying to acquire the lock.

The lower cost for releasing an uncontested lock explains the algorithm used to awaken waiters on a mutex. Historically, the mutex code would only awaken a single waiter when a contested lock was released, which left the lock in a contested state if there were more than one waiter. However, leaving a contested lock ensured that the new lock holder would have to perform a more expensive unlock operation. Indeed, all but the last waiter would have an expensive unlock operation. In the current FreeBSD system, all the waiters are awakened when the lock is released. Usually they end up being scheduled sequentially, which results in them all getting to do cheaper unlock operations. If they do all end up running concurrently, they will then use adaptive spinning and will finish the chain of lock requests sooner since the context switches to awaken the threads are performed in parallel rather than sequentially. This change in behavior was motivated by documentation of these effects noted in Solaris Internals [McDougall & Mauro, 2006].

It is wasteful of CPU cycles to use spin mutexes for resources that will be held for long periods of time (more than a few microseconds). For example, a spin mutex would be inappropriate for a disk buffer that would need to be locked throughout the time that a disk I/O was being done. Here, a sleep lock should be used. When a thread trying to acquire a medium- or long-term lock finds that the lock is held, it is put to sleep so that other threads can run until the lock becomes available.

Spin mutexes are never appropriate on a uniprocessor since the only way that a resource held by another thread will ever be released will be when that thread gets to run. Spin mutexes are always converted to sleep locks when running on a uniprocessor. As with the multi-processor, interrupts are disabled while the spin mutexes are held. Since there is no other processor on which the interrupts can run, interrupt latency becomes much more apparent on a uniprocessor.

Mutex Interface

The $mtx_init()$ function must be used to initialize a mutex before it can be used. The $mtx_init()$ function specifies a type that the witness code uses to classify a mutex when doing checks of lock ordering. It is not permissible to pass the same *mutex* to $mtx_init()$ multiple times without intervening calls to $mtx_destroy()$.

The $mtx_lock()$ function acquires a mutual exclusion lock for the currently running kernel thread. If another kernel thread is holding the mutex, the caller will sleep until the mutex is available. The $mtx_lock_spin()$ function is similar to the $mtx_lock()$ function except that it will spin until the mutex becomes available. A critical section is entered when the spin mutex is obtained and is exited when the spin mutex is released. Interrupts are blocked on the CPU on which the thread holding the spin mutex is running. No other threads, including interrupt threads, can run on the CPU during the period that the spin mutex is held.

It is possible for the same thread to acquire a mutex recursively with no ill effects if the MTX_RECURSE bit was passed to $mtx_init()$ during the initialization of the mutex. The witness module verifies that a thread does not recurse on a non-recursive lock. A recursive lock is useful if a resource may be locked at two or more levels in the kernel. By allowing a recursive lock, a lower layer need not check if the resource has already been locked by a higher layer; it can simply lock and release the resource as needed.

The *mtx_trylock()* function tries to acquire a mutual exclusion lock for the currently running kernel thread. If the mutex cannot be immediately acquired,

mtx_trylock() will return 0; otherwise the mutex will be acquired and a nonzero value will be returned. The *mtx_trylock()* function cannot be used with spin mutexes.

The *mtx_unlock()* function releases a mutual exclusion lock; if a higher-priority thread is waiting for the mutex, the releasing thread will be put to sleep to allow the higher-priority thread to acquire the mutex and run. A mutex that allows recursive locking maintains a reference count showing the number of times that it has been locked. Each successful lock request must have a corresponding unlock request. The mutex is not released until the final unlock has been done, causing the reference count to drop to zero.

The *mtx_unlock_spin()* function releases a spin-type mutual exclusion lock; the critical section entered before acquiring the mutex is exited.

The $mtx_destroy()$ function destroys a mutex so the data associated with it may be freed or otherwise overwritten. Any mutex that is destroyed must previously have been initialized with $mtx_init()$. It is permissible to have a single reference to a mutex when it is destroyed. It is not permissible to hold the mutex recursively or have another thread blocked on the mutex when it is destroyed. If these rules are violated, the kernel will panic.

Normally, a mutex is allocated within the structure that it will protect. For long-lived structures or structures that are allocated from a zone (structures in a zone are created once and used many times before they are destroyed), the time overhead of initializing and destroying it is insignificant. For a short-lived structure that is not allocated out of a zone, the cost of initializing and destroying an embedded mutex may exceed the time during which the structure is used. In addition, mutexes are large and may double or triple the size of a small short-lived structure (a mutex is often the size of a cache line, which is typically 128 bytes). To avoid this overhead, the kernel provides a pool of mutexes that may be borrowed for use with a short-lived structure. The short-lived structure does not need to reserve space for a mutex, just space for a pointer to a pool mutex. When the structure is allocated, it requests a pool mutex to which it sets its pointer. When it is done, the pool mutex is returned to the kernel and the structure freed. An example of a use of a pool mutex comes from the *poll* system call implementation that needs a structure to track a poll request from the time the system call is entered until the requested data arrives on the descriptor.

Lock Synchronization

Interprocess synchronization to a resource typically is implemented by associating it with a *lock* structure. The kernel has several lock managers that manipulate a lock. The operations provided by all the lock managers are:

- Request shared: Get one of many possible shared locks. If a thread holding an exclusive lock requests a shared lock, some lock managers will downgrade the exclusive lock to a shared lock while others simply return an error.
- Request exclusive: When all shared locks have cleared, grant an exclusive lock. To ensure that the exclusive lock will be granted quickly, some lock managers

stop granting shared locks when an exclusive lock is requested. Others grant new shared locks only for recursive lock requests. Only one exclusive lock may exist at a time, except that a thread holding an exclusive lock may get additional exclusive locks if the *canrecurse* flag was set when the lock was initialized. Some lock managers allow the *canrecurse* flag to be specified in the lock request.

• Request release: Release one instance of a lock.

In addition to these basic requests, some of the lock managers provide the following additional functions:

- Request upgrade: The thread must hold a shared lock that it wants to have upgraded to an exclusive lock. Other threads may get exclusive access to the resource between the time that the upgrade is requested and the time that it is granted. Some lock managers allow only a limited version of upgrade where it is granted if immediately available, but do not provide a mechanism to wait for an upgrade.
- Request exclusive upgrade: The thread must hold a shared lock that it wants to have upgraded to an exclusive lock. If the request succeeds, no other threads will have received exclusive access to the resource between the time that the upgrade is requested and the time that it is granted. However, if another thread has already requested an upgrade, the request will fail.
- Request downgrade: The thread must hold an exclusive lock that it wants to have downgraded to a shared lock. If the thread holds multiple (recursive) exclusive locks, some lock managers will downgrade them all to shared locks; other lock managers will fail the request.
- Request drain: Wait for all activity on the lock to end, and then mark it decommissioned. This feature is used before freeing a lock that is part of a piece of memory that is about to be released.

Locks must be initialized before their first use by calling their initialization function. Parameters to the initialization function may include the following:

- A top-half kernel priority at which the thread should run if it was blocked before it acquired the lock
- Flags such as *canrecurse* that allow the thread currently holding an exclusive lock to get another exclusive lock rather than panicking with a "locking against myself" failure
- A string that describes the resource that the lock protects, referred to as the *wait channel* message
- An optional maximum time to wait for the lock to become available

Not all types of locks support all these options. When a lock is no longer needed, it must be released.

As shown in Table 4.3, the lowest-level type of lock is the reader-writer lock. The reader-writer lock operates much like a mutex except that a reader-writer lock supports both shared and exclusive access. Like a mutex, it is managed by a *turnstile* so it cannot be held during a medium- or long-term sleep and provides priority propagation for exclusive (but not shared) locks. Reader-writer locks may be recursed.

Next up in Table 4.3 is the read-mostly lock. The read-mostly lock has the same capabilities and restrictions as reader-writer locks while they also add priority propagation for shared locks by tracking shared owners using a caller-supplied tracker data structure. Read-mostly locks are used to protect data that are read far more often than they are written. They work by trying the read without acquiring a lock assuming that the read will succeed and only fall back to using locks when the assumption fails. Reads usually happen more quickly but at a higher cost if the underlying resource is modified. The routing table is a good example of a read-mostly data structure. Routes are rarely updated, but are read frequently.

The remaining types of locks all permit medium- and long-term sleeping. None of these locks support priority propagation. The shared-exclusive locks are the fastest of these locks with the fewest features. In addition to the basic shared and exclusive access, they provide recursion for both shared and exclusive locks, the ability to be interrupted by a signal, and limited upgrade and downgrade capabilities.

The lock-manager locks are the most full featured but also the slowest of the locking schemes. In addition to the features of the shared-exclusive locks, they provide full upgrade and downgrade capabilities, the ability to be awakened after a specified interval, the ability to drain all users in preparation for being deallocated, and the ability to pass ownership of locks between threads and to the kernel.

Condition variables are used with mutexes to wait for conditions to occur. Threads wait on condition variables by calling $cv_wait()$, $cv_wait_sig()$ (wait unless interrupted by a signal), $cv_timedwait()$ (wait for a maximum time), or $cv_timedwait_sig()$ (wait unless interrupted by a signal or for a maximum time). Threads unblock waiters by calling $cv_signal()$ to unblock one waiter, or $cv_broadcast()$ to unblock all waiters. The $cv_waitq_remove()$ function removes a waiting thread from a condition-variable wait queue if it is on one.

A thread must hold a mutex before calling $cv_wait()$, $cv_wait_sig()$, $cv_timedwait()$, or $cv_timedwait_sig()$. When a thread waits on a condition, the mutex is atomically released before the thread is blocked, and then atomically reacquired before the function call returns. All waiters must use the same mutex with a condition variable. A thread must hold the mutex while calling $cv_signal()$ or $cv_broadcast()$.

Deadlock Prevention

The highest-level locking primitive prevents threads from deadlocking when locking multiple resources. Suppose that two threads, A and B, require exclusive access to two resources, R_1 and R_2 , to do some operation as shown in Figure 4.4. If thread A acquires R_1 and thread B acquires R_2 , then a deadlock occurs when



thread A tries to acquire R_2 and thread B tries to acquire R_1 . To avoid deadlock, FreeBSD maintains a partial ordering on all the locks. The two partial-ordering rules are as follows:

- 1. A thread may acquire only one lock in each class.
- 2. A thread may acquire only a lock in a higher-numbered class than the highestnumbered class for which it already holds a lock.

Figure 4.4 shows two classes. Class 1 with resources R_1 , R_1 , and R_1 . Class 2 with resources R_2 , R_2 , and R_2 . In Figure 4.4, Thread A holds R_1 and can request R_2 as R_1 and R_2 are in different classes and R_2 is in a higher-numbered class than R_1 . However, Thread B must release R_2 before requesting R_1 , since R_2 is in a higher class than R_1 . Thus, Thread A will be able to acquire R_2 when it is released by Thread B. After Thread A completes and releases R_1 and R_2 , Thread B will be able to acquire both of those locks and run to completion without deadlock.

Historically, the class members and ordering were poorly documented and unenforced. Violations were discovered when threads would deadlock and a careful analysis was done to figure out what ordering had been violated. With an increasing number of developers and a growing kernel, the ad hoc method of maintaining the partial ordering of locks became untenable. A witness module was added to the kernel to derive and enforce the partial ordering of the locks. The witness module keeps track of the locks acquired and released by each thread. It also keeps track of the order in which locks are acquired relative to each other. Each time a lock is acquired, the witness module uses these two lists to verify that a lock is not being acquired in the wrong order. If a lock order violation is detected, then a message is output to the console detailing the locks involved and the locations in the code in which they were acquired. The witness module also verifies that no locks that prohibit sleeping are held when requesting a sleep lock or voluntarily going to sleep. The witness module can be configured to either panic or drop into the kernel debugger when an order violation occurs or some other witness check fails. When running the debugger, the witness module can output the list of locks held by the current thread to the console along with the filename and line number at which each lock was last acquired. It can also dump the current order list to the console. The code first displays the lock order tree for all the sleep locks. Then it displays the lock order tree for all the sleep locks that have not yet been acquired.

4.4 Thread Scheduling

The FreeBSD scheduler has a well-defined set of kernel-application programming interfaces (kernel APIs) that allow it to support different schedulers. Since FreeBSD 5.0, the kernel has had two schedulers available:

- The ULE scheduler first introduced in FreeBSD 5.0 and found in the file /sys/kern/sched_ule.c [Roberson, 2003]. The name is not an acronym. If the underscore in its filename is removed, the rationale for its name becomes apparent. This scheduler is used by default and is described later in this section.
- The traditional 4.4BSD scheduler found in the file /sys/kern/sched_4bsd.c. This scheduler is still maintained but no longer used by default.

Because a busy system makes millions of scheduling decisions per second, the speed with which scheduling decisions are made is critical to the performance of the system as a whole. Other UNIX systems have added a dynamic scheduler switch that must be traversed for every scheduling decision. To avoid this overhead, FreeBSD requires that the scheduler be selected at the time the kernel is built. Thus, all calls into the scheduling code are resolved at compile time rather than going through the overhead of an indirect function call for every scheduling decision.

The Low-Level Scheduler

Scheduling is divided into two parts: a simple low-level scheduler that runs frequently and a more complex high-level scheduler that runs at most a few times per second. The low-level scheduler runs every time a thread blocks and a new thread must be selected to run. For efficiency when running thousands of times per second, it must make its decision quickly with a minimal amount of information. To simplify its task, the kernel maintains a set of *run queue*s for each CPU in the system that are organized from high to low priority. When a task blocks on a CPU, the low-level scheduler's sole responsibility is to select the thread from the highest-priority non-empty run queue for that CPU. The high-level scheduler is responsible for setting the thread priorities and deciding on which CPU's run queue they should be placed. Each CPU has its own set of run queues to avoid contention for access when two CPUs both need to select a new thread to run at the same time. Contention between run queues occurs only when the high-level scheduler decides to move a thread from the run queue of one CPU to the run queue of another CPU. The kernel tries to avoid moving threads between CPUs as the loss of its CPU-local caches slows it down.

All threads that are runnable are assigned a scheduling priority and a CPU by the high-level scheduler that determines in which run queue they are placed. In selecting a new thread to run, the low-level scheduler scans the run queues of the CPU needing a new thread from highest to lowest priority and chooses the first thread on the first nonempty queue. If multiple threads reside on a queue, the system runs them *round robin*; that is, it runs them in the order that they are found on the queue, with equal amounts of time allowed. If a thread blocks, it is not put back onto any run queue. Instead, it is placed on a *turnstile* or a *sleepqueue*. If a thread uses up the *time quantum* (or *time slice*) allowed it, it is placed at the end of the queue from which it came, and the thread at the front of the queue is selected to run.

The shorter the time quantum, the better the interactive response. However, longer time quanta provide higher system throughput because the system will incur less overhead from doing context switches and processor caches will be flushed less often. The time quantum used by FreeBSD is adjusted by the high-level scheduler as described later in this subsection.

Thread Run Queues and Context Switching

The kernel has a single set of run queues to manage all the thread scheduling classes shown in Table 4.2. The scheduling-priority calculations described in the previous section are used to order the set of timesharing threads into the priority ranges between 120 and 223. The real-time threads and the idle threads priorities are set by the applications themselves but are constrained by the kernel to be within the ranges 48 to 79 and 224 to 255, respectively. The number of queues used to hold the collection of all runnable threads in the system affects the cost of managing the queues. If only a single (ordered) queue is maintained, then selecting the next runnable thread becomes simple but other operations become expensive. Using 256 different queues can significantly increase the cost of identifying the next thread to run. The system uses 64 run queues, selecting a run queue for a thread by dividing the thread's priority by 4. To save time, the threads on each queue are not further sorted by their priorities.

The run queues contain all the runnable threads in main memory except the currently running thread. Figure 4.5 shows how each queue is organized as a doubly linked list of thread structures. The head of each run queue is kept in an array. Associated with this array is a bit vector, rq_status , that is used in identifying the nonempty run queues. Two routines, $runq_add()$ and $runq_remove()$, are used to place a thread at the tail of a run queue, and to take a thread off the head of a run queue. The heart of the scheduling algorithm is the $runq_choose()$ routine. The $runq_choose()$ routine is responsible for selecting a new thread to run; it operates as follows:



1. Ensures that our caller acquired the lock associated with the run queue.

- Locates a nonempty run queue by finding the location of the first nonzero bit in the rq_status bit vector. If rq_status is zero, there are no threads to run, so selects an *idle loop* thread.
- 3. Given a nonempty run queue, removes the first thread on the queue.
- 4. If this run queue is now empty as a result of removing the thread, clears the appropriate bit in *rq_status*.
- 5. Returns the selected thread.

The context-switch code is broken into two parts. The machine-independent code resides in $mi_switch()$; the machine-dependent part resides in $cpu_switch()$. On most architectures, $cpu_switch()$ is coded in assembly language for efficiency.

Given the *mi_switch()* routine and the thread-priority calculations, the only missing piece in the scheduling facility is how the system forces an involuntary context switch. Remember that voluntary context switches occur when a thread calls the *sleep()* routine. *Sleep()* can be invoked only by a runnable thread, so *sleep()* needs only to place the thread on a sleep queue and to invoke *mi_switch()* to schedule the next thread to run. Often, an interrupt thread will not want to *sleep()* itself but will be delivering data that will cause the kernel to want to run a different thread than the one that was running before the interrupt. Thus, the kernel needs a mechanism to request that an involuntary context switch be done at the conclusion of the interrupt.

This mechanism is handled by setting the currently running thread's TDF_NEEDRESCHED flag and then posting an *asynchronous system trap* (*AST*). An AST is a trap that is delivered to a thread the next time that thread is preparing to return from an interrupt, a trap, or a system call. Some architectures

support ASTs directly in hardware; other systems emulate ASTs by checking an AST flag at the end of every system call, trap, and interrupt. When the hardware AST trap occurs or the AST flag is set, the *mi_switch()* routine is called instead of the current thread resuming execution. Rescheduling requests are made by the *sched_lend_user_prio()*, *sched_clock()*, *sched_setpreempt()*, and *sched_affinity()* routines.

With the advent of multiprocessor support, FreeBSD can preempt threads executing in kernel mode. However, such preemption is generally not done for threads running in the timesharing class, so the worst-case real-time response to events when running with the timeshare scheduler is defined by the longest path through the top half of the kernel. Since the system guarantees no upper bounds on the duration of a system call, when running with just the timeshare scheduler FreeBSD is decidedly not a hard real-time system.

Real-time and interrupt threads do preempt lower-priority threads. The longest path that preemption is disabled for real-time and interrupt threads is defined by the longest time a spinlock is held or a critical section is entered. Thus, when using real-time threads, microsecond real-time deadlines can be met. The kernel can be configured to preempt timeshare threads executing in the kernel with other higher-priority timeshare threads. This option is not used by default as the increase in context switches adds overhead and does not help make timeshare threads response time more predictable.

Timeshare Thread Scheduling

The goal of a multiprocessing system is to apply the power of multiple CPUs to a problem, or set of problems, to achieve a result in less time than it would run on a single-processor system. If a system has the same number of runnable threads as it does CPUs, then achieving this goal is easy. Each runnable thread gets a CPU to itself and runs to completion. Typically, there are many runnable threads competing for a few processors. One job of the scheduler is to ensure that the CPUs are always busy and are not wasting their cycles. When a thread completes its work, or is blocked waiting for resources, it is removed from the processor on which it was running. While a thread is running on a processor, it brings its working setthe instructions it is executing and the data on which it is operating-into the CPU's memory cache. Migrating a thread has a cost. When a thread is moved from one CPU to another, its CPU-cache working set is lost and must be removed from the CPU on which it was running and then loaded into the new CPU to which it has been migrated. The performance of a multiprocessing system with a naive scheduler that does not take this cost into account can fall beneath that of a singleprocessor system. The term processor affinity describes a scheduler that only migrates threads when necessary to give an idle processor something to do.

A multiprocessing system may be built with multiple processor chips. Each processor chip may have multiple CPU cores, each of which can execute a thread. The CPU cores on a single processor chip share many of the processor's resources, such as memory caches and access to main memory, so they are more tightly synchronized than the CPUs on other processor chips.

Handling processor chips with multiple CPUs is a derivative form of load balancing among CPUs on different chips. It is handled by maintaining a hierarchy of CPUs. The CPUs on the same chip are the cheapest between which to migrate threads. Next down in the hierarchy are processor chips on the same motherboard. Below them are chips connected by the same backplane. The scheduler supports an arbitrary depth hierarchy as dictated by the hardware. When the scheduler is deciding to which processor to migrate a thread, it will try to pick a new processor higher in the hierarchy because that is the lowest-cost migration path.

From a thread's perspective, it does not know that there are other threads running on the same processor because the processor is handling them independently. The one piece of code in the system that needs to be aware of the multiple CPUs is the scheduling algorithm. In particular, the scheduler treats each CPU on a chip as one on which it is cheaper to migrate threads than it would be to migrate the thread to a CPU on another chip. The mechanism for getting tighter affinity between CPUs on the same processor chip versus CPUs on other processor chips is described later in this section.

The traditional FreeBSD scheduler maintains a global list of runnable threads that it traverses once per second to recalculate their priorities. The use of a single list for all runnable threads means that the performance of the scheduler is dependent on the number of tasks in the system, and as the number of tasks grow, more CPU time must be spent in the scheduler maintaining the list.

The ULE scheduler was developed during FreeBSD 5.0 with major work continuing into FreeBSD 9.0, spanning 10 years of development. The scheduler was developed to address shortcomings of the traditional BSD scheduler on multiprocessor systems. A new scheduler was undertaken for several reasons:

- To address the need for processor affinity in multiprocessor systems
- To supply equitable distribution of load between CPUs on multiprocessor systems
- To provide better support for processors with multiple CPU cores on a single chip
- To improve the performance of the scheduling algorithm so that it is no longer dependent on the number of threads in the system
- To provide interactivity and timesharing performance similar to the traditional BSD scheduler.

The traditional BSD scheduler had good interactivity on large timeshare systems and single-user desktop and laptop systems. However, it had a single global run queue and consequently a single global scheduler lock. Having a single global run queue was slowed both by contention for the global lock and by difficulties implementing CPU affinity.

The priority computation relied on a single global timer that iterated over every runnable thread in the system and evaluated its priority while holding several highly contended locks. This approach became slower as the number of runnable threads increased. While the priority calculations were being done, processes could not *fork* or *exit* and CPUs could not context switch.

The ULE scheduler can logically be thought of as two largely orthogonal sets of algorithms; those that manage the affinity and distribution of threads among CPUs and those that are responsible for the order and duration of a thread's runtime. These two sets of algorithms work in concert to provide a balance of low latency, high throughput, and good resource utilization. The remainder of the scheduler is event driven and uses these algorithms to implement various decisions according to changes in system state.

The goal of equalling the exceptional interactive behavior and throughput of the traditional BSD scheduler in a multiprocessor-friendly and constant-time implementation was the most challenging and time consuming part of ULE's development. The interactivity, CPU utilization estimation, priority, and time slice algorithms together implement the timeshare scheduling policy.

The behavior of threads is evaluated by ULE on an event-driven basis to differentiate interactive and batch threads. Interactive threads are those that are thought to be waiting for and responding to user input. They require low latency to achieve a good user experience. Batch threads are those that tend to consume as much CPU as they are given and may be background jobs. A good example of the former is a text editor, and for the latter, a compiler. The scheduler must use imperfect heuristics to provide a gradient of behaviors based on a best guess of the category to which a given thread fits. This categorization may change frequently during the lifetime of a thread and must be responsive on timescales relevant to people using the system.

The algorithm that evaluates interactivity is called the interactivity score. The interactivity score is the ratio of voluntary sleep time to run time normalized to a number between 0 and 100. This score does not include time waiting on the run queue while the thread is not yet the highest priority thread in the queue. By requiring explicit voluntary sleeps, we can differentiate threads that are not running because of inferior priority versus those that are periodically waiting for user input. This requirement also makes it more challenging for a thread to be marked interactive as system load increases, which is desirable because it prevents the system from becoming swamped with interactive threads while keeping things like shells and simple text editors available to administrators. When plotted, the interactivity scores derived from a matrix of possible sleep and run times becomes a three-dimensional sigmoid function. Using this approach means that interactive tasks tend to stay interactive and batch tasks tend to stay batched.

A particular challenge is complex X Window applications such as Web browsers and office productivity packages. These applications may consume significant resources for brief periods of time, however the user expects them to remain interactive. To resolve this issue, a several-second history of the sleep and run behavior is kept and gradually decayed. Thus, the scheduler keeps a moving average that can tolerate bursts of behavior but will quickly penalize timeshare threads that abuse their elevated status. A longer history allows longer bursts but learns more slowly. The interactivity score is compared to the interactivity threshold, which is the cutoff point for considering a thread interactive. The interactivity threshold is modified by the process *nice* value. Positive nice values make it more challenging for a thread to be considered interactive, while negative values make it easier. Thus, the nice value gives the user some control over the primary mechanism of reducing thread-scheduling latency.

A thread is considered to be interactive if the ratio of its voluntary sleep time versus its run time is below a certain threshold. The interactivity threshold is defined in the ULE code and is not configurable. ULE uses two equations to compute the interactivity score of a thread. For threads whose sleep time exceeds their run time, Eq 4.1 is used:

interactivity score =
$$\frac{scaling \ factor}{sleep \ / \ run}$$
 (Eq. 4.1)

When a thread's run time exceeds its sleep time, Eq. 4.2 is used instead:

interactivity score =
$$\frac{scaling \ factor}{run \ / \ sleep} + scaling \ factor$$
 (Eq. 4.2)

The scaling factor is the maximum interactivity score divided by two. Threads that score below the interactivity threshold are considered to be interactive; all others are noninteractive. The *sched_interact_update()* routine is called at several points in a threads existence—for example, when the thread is awakened by a *wakeup()* call—to update the thread's run time and sleep time. The sleep- and run-time values are only allowed to grow to a certain limit. When the sum of the run time and sleep time pass the limit, they are reduced to bring them back into range. An interactive thread whose sleep history was not remembered at all would not remain interactive, resulting in a poor user experience. Remembering an interactive thread's sleep time for too long would allow the thread to get more than its fair share of the CPU. The amount of history that is kept and the interactivity threshold are the two values that most strongly influence a user's interactive experience on the system.

Priorities are assigned according to the thread's interactivity status. Interactive threads have a priority that is derived from the interactivity score and are placed in a priority band above batch threads. They are scheduled like real-time round-robin threads. Batch threads have their priorities determined by the estimated CPU utilization modified according to their process nice value. In both cases, the available priority range is equally divided among possible interactive scores or percent-cpu calculations, both of which are values between 0 and 100. Since there are fewer than 100 priorities available for each class, some values share priorities. Both computations roughly assign priorities according to a history of CPU utilization but with different longevities and scaling factors.

The CPU utilization estimator accumulates run time as a thread runs and decays it as a thread sleeps. The utilization estimator provides the percent-cpu values displayed in **top** and **ps**. ULE delays the decay until a thread wakes to avoid periodically scanning every thread in the system. Since this delay leaves

values unchanged for the duration of sleeps, the values must also be decayed before any user process inspects them. This approach preserves the constant-time and event-driven nature of the scheduler.

The CPU utilization is recorded in the thread as the number of ticks, typically 1 millisecond, during which a thread has been running, along with window of time defined as a first and last tick. The scheduler attempts to keep roughly 10 seconds of history. To accomplish decay, it waits until there are 11 seconds of history and then subtracts one-tenth of the tick value while moving the first tick forward 1 second. This inexpensive, estimated moving-average algorithm has the property of allowing arbitrary update intervals. If the utilization information is inspected after more than the update interval has passed, the tick value is zeroed. Otherwise, the number of seconds that have passed divided by the update interval is subtracted.

The scheduler implements round-robin through the assignment of time slices. A time slice is a fixed interval of allowed run time before the scheduler will select another thread of equal priority to run. The time slice prevents starvation among equal priority threads. The time slice times the number of runnable threads in a given priority defines the maximum latency a thread of that priority will experience before it can run. To bound this latency, ULE dynamically adjusts the size of slices it dispenses based on system load. The time slice has a minimum value to prevent thrashing and balance throughput with latency. An interrupt handler calls the scheduler to evaluate the time slice during every statclock tick. Using the statclock to evaluate the time slice is a stochastic approach to slice accounting that is efficient but only grossly accurate.

The scheduler must also work to prevent starvation of low-priority batch jobs by higher-priority batch jobs. The traditional BSD scheduler avoided starvation by periodically iterating over all threads waiting on the run queue to elevate the lowpriority threads and decrease the priority of higher-priority threads that had been monopolizing the CPU. This algorithm violates the desire to run in constant time independent of the number of system threads. As a result, the run queue for batch-policy timeshare threads is kept in a similar fashion to the system callwheel, also known as a calendar queue. A calendar queue is one in which the queue's head and tail rotate according to a clock or period. An element can be inserted into a calendar queue many positions away from the head and gradually migrate toward the head. Because this run queue is special purpose, it is kept separately from the real-time and idle queues while interactive threads are kept along with the real-time threads until they are no longer considered interactive.

The ULE scheduler creates a set of three arrays of queues for each CPU in the system. Having per-CPU queues makes it possible to implement processor affinity in a multiprocessor system.

One array of queues is the *idle queue*, where all idle threads are stored. The array is arranged from highest to lowest priority. The second array of queues is designated the realtime queue. Like the idle queue, it is arranged from highest to lowest priority.

The third array of queues is designated the timeshare queue. Rather than being arranged in priority order, the timeshare queues are managed as a calendar queue. A pointer references the current entry. The pointer is advanced once per system tick, although it may not advance on a tick until the currently selected queue is empty. Since each thread is given a maximum time slice and no threads may be added to the current position, the queue will drain in a bounded amount of time. This requirement to empty the queue before advancing to the next queue means that the wait time a thread experiences is not only a function of its priority but also the system load.

Insertion into the timeshare queue is defined by the relative difference between a thread's priority and the best possible timeshare priority. High-priority threads will be placed soon after the current position. Low-priority threads will be placed far from the current position. This algorithm ensures that even the lowest-priority timeshare thread will eventually make it to the selected queue and execute in spite of higher-priority timeshare threads being available in other queues. The difference in priorities of two threads will determine their ratio of run-time. The higher-priority thread may be inserted ahead of the lower-priority thread multiple times before the queue position catches up. This run-time ratio is what grants timeshare CPU hogs with different nice values, different proportional shares of the CPU.

These algorithms taken together determine the priorities and run times of timesharing threads. They implement a dynamic tradeoff between latency and throughput based on system load, thread behavior, and a range of effects based on user-scheduling decisions made with nice. Many of the parameters governing the limits of these algorithms can be explored in real time with the *sysctl kern.sched* tree. The rest are compile-time constants that are documented at the top of the scheduler source file (/**sys/kern/sched_ule.c**).

Threads are picked to run, in priority order, from the realtime queue until it is empty, at which point threads from the currently selected timeshare queue will be run. Threads in the idle queues are run only when the other two arrays of queues are empty. Real-time and interrupt threads are always inserted into the realtime queues so that they will have the least possible scheduling latency. Interactive threads are also inserted into the realtime queue to keep the interactive response of the system acceptable.

Noninteractive threads are put into the timeshare queues and are scheduled to run when the queues are switched. Switching the queues guarantees that a thread gets to run at least once every pass around the array of the timeshare queues regardless of priority, thus ensuring fair sharing of the processor.

Multiprocessor Scheduling

A principal goal behind the development of ULE was improving performance on multiprocessor systems. Good multiprocessing performance involves balancing affinity with utilization and the preservation of the illusion of global scheduling in a system with local scheduling queues. These decisions are implemented using a CPU topology supplied by machine-dependent code that describes the relationships between CPUs in the system. The state is evaluated whenever a thread becomes runnable, a CPU idles, or a periodic task runs to rebalance the load.

These events form the entirety of the multiprocessor-aware scheduling decisions.

The topology system was devised to identify which CPUs were symmetric multi-threading peers and then made generic to support other relationships. Some examples are CPUs within a package, CPUs sharing a layer of cache, CPUs that are local to a particular memory, or CPUs that share execution units such as in symmetric multi-threading. This topology is implemented as a tree of arbitrary depth where each level describes some shared resource with a cost value and a bitmask of CPUs sharing that resource. The root of the tree holds CPUs in a system with branches to each socket, then shared cache, shared functional unit, etc. Since the system is generic, it should be extensible to describe any future processor arrangement. There is no restriction on the depth of the tree or requirement that all levels are implemented.

Parsing this topology is a single recursive function called $cpu_search()$. It is a path-aware, goal-based, tree-traversal function that may be started from arbitrary subtrees. It may be asked to find the least- or most-loaded CPU that meets a given criteria, such as a priority or load threshold. When considering load, it will consider the load of the entire path, thus giving the potential for balancing sockets, caches, chips, etc. This function is used as the basis for all multiprocessingrelated scheduling decisions. Typically, recursive functions are avoided in kernel programming because there is potential for stack exhaustion. However, the depth is fixed by the depth of the processor topology that typically does not exceed three.

When a thread becomes runnable as a result of a wakeup, unlock, thread creation, or other event, the *sched_pickcpu()* function is called to decide where it will run. ULE determines the best CPU based on the following criteria:

- Threads with hard affinity to a single CPU or short-term binding pick the only allowed CPU.
- Interrupt threads that are being scheduled by their hardware interrupt handlers are scheduled on the current CPU if their priority is high enough to run immediately.
- Thread affinity is evaluated by walking backwards up the tree starting from the last CPU on which it was scheduled until a package or CPU is found with valid affinity that can run the thread immediately.
- The whole system is searched for the least-loaded CPU that is running a lowerpriority thread than the one to be scheduled.
- The whole system is searched for the least-loaded CPU.
- The results of these searches are compared to the current CPU to see if that would give a preferable decision to improve locality among the sleeping and waking threads as they may share some state.

This approach orders from most preferential to least preferential. The affinity is valid if the sleep time of the thread was shorter than the product of a time constant and a largest-cache-shared level in the topology. This computation coarsely models the time required to push state out of the cache. Each thread has a bitmap of allowed CPUs that is manipulated by **cpuset** and is passed to $cpu_search()$ for every decision. The locality between sleeper and waker can improve producer/consumer type threading situations when they have shared cache state but it can also cause underutilization when each thread would run faster given its own CPU. These examples exemplify the types of decisions that must be made with imperfect information.

The next major multiprocessing algorithm runs when a CPU idles. The CPU sets a bit in a bitmask shared by all processors that says that it is idle. The idle CPU calls *tdq_idled()* to search other CPUs for work that can be migrated, or stolen in ULE terms, to keep the CPU busy. To avoid thrashing and excessive migration, the kernel sets a load threshold that must be exceeded on another CPU before some load will be taken. If any CPU exceeds this threshold, the idle CPU will search its run queues for work to migrate. The highest-priority work that can be scheduled on the idle CPU is then taken. This migration may be detrimental to affinity but improves many latency-sensitive workloads.

Work may also be pushed to an idle CPU. Whenever an active CPU is about to add work to its own run queue, it first checks to see if it has excess work and if another CPU in the system is idle. If an idle CPU is found, then the thread is migrated to the idle CPU using an *interprocessor interrupt (IPI)*. Making a migration decision by inspecting a shared bitmask is much faster than scanning the run queues of all the other processors. Seeking out idle processors when adding a new task works well because it spreads the load when it is presented to the system.

The last major multiprocessing algorithm is the long-term load balancer. This form of migration, called *push migration*, is done by the system on a periodic basis and more aggressively offloads work to other processors in the system. Since the two scheduling events that distribute load only run when a thread is added and when a CPU idles, it is possible to have a long-term imbalance where more threads are running on one CPU than another. Push migration ensures fairness among the runnable threads. For example, with three runnable threads on a two-processor system, it would be unfair for one thread to get a processor to itself while the other two had to share the second processor. To fulfill the goal of emulating a fair global run queue, ULE must periodically shuffle threads to keep the system balanced. By pushing a thread from the processor with two threads to the processor with one thread, no single thread would get to run alone indefinitely. An ideal implementation would give each thread an average of 66 percent of the CPU available from a single CPU.

The long-term load balancer balances the worst path pair in the hierarchy to avoid socket-, cache-, and chip-level imbalances. It runs from an interrupt handler in a randomized interval of roughly 1 second. The interval is randomized to prevent harmonic relationships between periodic threads and the periodic load balancer. In much the same way a stochastic sampling profiler works, the balancer picks the most- and least-loaded path from the current tree position and then recursively balances those paths by migrating threads. The scheduler must decide whether it is necessary to send an IPI when adding a thread to a remote CPU, just as it must decide whether adding a thread to the current CPU should preempt the current thread. The decision is made based on the current priority of the thread running on the target CPU and the priority of the thread being scheduled. Preempting whenever the pushed thread has a higher priority than the currently running thread results in excessive interrupts and preemptions. Thus, a thread must exceed the timesharing priority before an IPI is generated. This requirement trades some latency in batch jobs for improved performance.

A notable omission to the load balancing events is thread preemption. Preempted threads are simply added back to the run queue of the current CPU. An additional load-balancing decision can be made here. However, the runtime of the preempting thread is not known and the preempted thread may maintain affinity. The scheduler optimistically chooses to wait and assume affinity is more valuable than latency.

Each CPU in the system has its own set of run queues, statistics, and a lock to protect these fields in a *thread-queue* structure. During migration or a remote wakeup, a lock may be acquired by a CPU other than the one owning the queue. In practice, contention on these locks is rare unless the workload exhibits grossly overactive context switching and thread migration, typically suggesting a higher-level problem. Whenever a pair of these locks is required, such as for load balancing, a special function locks the pair with a defined lock order. The lock order is the lock with the lowest pointer value first. These per-CPU locks and queues resulted in nearly linear scaling with well-behaved workloads in cases where performance previously did not improve with the addition of new CPUs and occasion-ally have decreased as new CPUs introduced more contention. The design has scaled well from single CPUs to 512-thread network processors.

Adaptive Idle

Many workloads feature frequent interrupts that do little work but need low latency. These workloads are common in low-throughput, high-packet-rate net-working. For these workloads, the cost of waking the CPU from a low-power state, possibly with an IPI from another CPU, is excessive. To improve performance, ULE includes a feature that optimistically spins, waiting for load when the CPU has been context switching at a rate exceeding a set frequency. When this frequency lowers or we exceed the adaptive spin count, the CPU is put into a deeper sleep.

Traditional Timeshare Thread Scheduling

The traditional FreeBSD timeshare-scheduling algorithm is based on *multilevel feedback queues*. The system adjusts the priority of a thread dynamically to reflect resource requirements (e.g., being blocked awaiting an event) and the amount of resources consumed by the thread (e.g., CPU time). Threads are moved between run queues based on changes in their scheduling priority (hence the word

"feedback" in the name *multilevel feedback queue*). When a thread other than the currently running thread attains a higher priority (by having that priority either assigned or given when it is awakened), the system switches to that thread immediately if the current thread is in user mode. Otherwise, the system switches to the higher-priority thread as soon as the current thread exits the kernel. The system tailors this *short-term-scheduling algorithm* to favor interactive jobs by raising the scheduling priority of threads that are blocked waiting for I/O for 1 or more seconds and by lowering the priority of threads that accumulate significant amounts of CPU time.

The time quantum is always 0.1 second. This value was empirically found to be the longest quantum that could be used without loss of the desired response for interactive jobs such as editors. Perhaps surprisingly, the time quantum remained unchanged over the 30-year lifetime of this scheduler. Although the time quantum was originally selected on centralized timesharing systems with many users, it has remained correct for decentralized laptops. While laptop users expect a response time faster than that anticipated by the original timesharing users, the shorter run queues on the single-user laptop made a shorter quantum unnecessary.

4.5 Process Creation

In FreeBSD, new processes are created with the *fork* family of system calls. The *fork* system call creates a complete copy of the parent process. The *rfork* system call creates a new process entry that shares a selected set of resources from its parent rather than making copies of everything. The *vfork* system call differs from *fork* in how the virtual-memory resources are treated; *vfork* also ensures that the parent will not run until the child does either an *exec* or *exit* system call. The *vfork* system call is described in Section 6.6.

The process created by a *fork* is termed a *child process* of the original *parent process*. From a user's point of view, the child process is an exact duplicate of the parent process except for two values: the child PID and the parent PID. A call to *fork* returns the child PID to the parent and zero to the child process. Thus, a program can identify whether it is the parent or child process after a *fork* by checking this return value.

A fork involves three main steps:

- 1. Allocating and initializing a new process structure for the child process
- 2. Duplicating the context of the parent (including the thread structure and virtualmemory resources) for the child process
- 3. Scheduling the child process to run

The second step is intimately related to the operation of the memory-management facilities described in Chapter 6. Consequently, only those actions related to process management will be described here.

The kernel begins by allocating memory for the new process and thread entries (see Figure 4.1). These thread and process entries are initialized in three steps: One part is copied from the parent's corresponding structure, another part is zeroed, and the rest is explicitly initialized. The zeroed fields include recent CPU utilization, wait channel, swap and sleep time, timers, tracing, and pending-signal information. The copied portions include all the privileges and limitations inherited from the parent, including the following:

- The process group and session
- The signal state (ignored, caught, and blocked signal masks)
- The *p_nice* scheduling parameter
- A reference to the parent's credential
- A reference to the parent's set of open files
- A reference to the parent's limits

The child's explicitly set information includes:

- The process's signal-actions structure
- · Zeroing the process's statistics structure
- Entry onto the list of all processes
- Entry onto the child list of the parent and the back pointer to the parent
- Entry onto the parent's process-group list
- Entry onto the hash structure that allows the process to be looked up by its PID
- A new PID for the process

The new PID must be unique among all processes. Early versions of BSD verified the uniqueness of a PID by performing a linear search of the process table. This search became infeasible on large systems with many processes. FreeBSD maintains a range of unallocated PIDs between *lastpid* and *pidchecked*. It allocates a new PID by incrementing and then using the value of *lastpid*. When the newly selected PID reaches *pidchecked*, the system calculates a new range of unused PIDs by making a single scan of all existing processes (not just the active ones are scanned—zombie and swapped processes also are checked).

The final step is to copy the parent's address space. To duplicate a process's image, the kernel invokes the memory-management facilities through a call to $vm_forkproc()$. The $vm_forkproc()$ routine is passed a pointer to the initialized process structure for the child process and is expected to allocate all the resources that the child will need to execute. The call to $vm_forkproc()$ returns through a different execution path directly into user mode in the child process and via the normal execution path in the parent process.

Once the child process is fully built, its thread is made known to the scheduler by being placed on the run queue. The alternate return path will set the return value of *fork* system call in the child to 0. The normal execution return path in the parent sets the return value of the *fork* system call to be the new PID.

4.6 Process Termination

Processes terminate either voluntarily through an *exit* system call or involuntarily as the result of a signal. In either case, process termination causes a status code to be returned to the parent of the terminating process (if the parent still exists). This termination status is returned through the *wait4* system call. The *wait4* call permits an application to request the status of both stopped and terminated processes. The *wait4* request can wait for any direct child of the parent, or it can wait selectively for a single child process or for only its children in a particular process group. *Wait4* can also request statistics describing the resource utilization of a terminated child process. Finally, the *wait4* interface allows a process to request status codes without blocking.

Within the kernel, a process terminates by calling the *exit()* routine. The *exit()* routine first kills off any other threads associated with the process. The termination of other threads is done as follows:

- Any thread entering the kernel from userspace will *thread_exit()* when it traps into the kernel.
- Any thread already in the kernel and attempting to sleep will return immediately with EINTR or EAGAIN, which will force them back out to userspace, freeing resources as they go. When the thread attempts to return to userspace, it will instead hit *exit()*.

The *exit()* routine then cleans up the process's kernel-mode execution state by doing the following:

- Canceling any pending timers
- · Releasing virtual-memory resources
- Closing open descriptors
- · Handling stopped or traced child processes

With the kernel-mode state reset, the process is then removed from the list of active processes—the *allproc* list—and is placed on the list of *zombie processes* pointed to by *zombproc*. The process state is changed to show that no thread is currently running. The *exit()* routine then does the following:
- Records the termination status in the *p_xstat* field of the process structure
- Bundles up a copy of the process's accumulated resource usage (for accounting purposes) and hangs this structure from the *p_ru* field of the process structure
- Notifies the deceased process's parent

Finally, after the parent has been notified, the *cpu_exit()* routine frees any machine-dependent process resources and arranges for a final context switch from the process.

The *wait4* call works by searching a process's descendant processes for ones that have entered the ZOMBIE state (e.g., that have terminated). If a process in ZOMBIE state is found that matches the wait criterion, the system will copy the termination status from the deceased process. The process entry then is taken off the zombie list and is freed. Note that resources used by children of a process are accumulated only as a result of a *wait4* system call. When users are trying to analyze the behavior of a long-running program, they will find it useful to be able to obtain this resource usage information before the termination of a process. Although the information is available inside the kernel and within the context of that program, there is no interface to request it outside that context until process termination.

4.7 Signals

Signals were originally designed to model exceptional events, such as an attempt by a user to kill a runaway program. They were not intended to be used as a general *interprocess-communication* mechanism, and thus no attempt was made to make them reliable. In earlier systems, whenever a signal was caught, its action was reset to the default action. The introduction of job control brought much more frequent use of signals and made more visible a problem that faster processors also exacerbated: If two signals were sent rapidly, the second could cause the process to die, even though a signal handler had been set up to catch the first signal. At this time, reliability became desirable, so the developers designed a new framework that contained the old capabilities as a subset while accommodating new mechanisms.

The signal facilities found in FreeBSD are designed around a *virtual-machine* model, in which system calls are considered to be the parallel of a machine's hardware instruction set. Signals are the software equivalent of traps or interrupts, and signal-handling routines do the equivalent function of interrupt or trap service routines. Just as machines provide a mechanism for blocking hardware interrupts so that consistent access to data structures can be ensured, the signal facilities allow software signals to be masked. Finally, because complex run-time stack environments may be required, signals, like interrupts, may be handled on an alternate application-provided run-time stack. These machine models are summarized in Table 4.4

Hardware Machine	Software Virtual Machine
instruction set	set of system calls
restartable instructions	restartable system calls
interrupts/traps	signals
interrupt/trap handlers	signal handlers
blocking interrupts	masking signals
interrupt stack	signal stack

Table 4.4 Comparison of hardware-machine operations and the corresponding software virtual-machine operations.

FreeBSD defines a set of *signals* for software and hardware conditions that may arise during the normal execution of a program; these signals are listed in Table 4.5. Signals may be delivered to a process through application-specified *signal handlers* or may result in default actions, such as process termination, carried out by the system. FreeBSD signals are designed to be software equivalents of hardware interrupts or traps.

Each signal has an associated action that defines how it should be handled when it is delivered to a process. If a process contains more than one thread, each thread may specify whether it wishes to take action for each signal. Typically, one thread elects to handle all the process-related signals such as interrupt, stop, and continue. All the other threads in the process request that the process-related signals be masked out. Thread-specific signals such as segmentation fault, floating point exception, and illegal instruction are handled by the thread that caused them. Thus, all threads typically elect to receive these signals. The precise disposition of signals to threads is given in the later subsection on posting a signal. First, we describe the possible actions that can be requested.

The disposition of signals is specified on a per-process basis. If a process has not specified an action for a signal, it is given a default action (see Table 4.5) that may be any one of the following:

- · Ignoring the signal
- · Terminating all the threads in the process
- Terminating all the threads in the process after generating a *core file* that contains the process's execution state at the time the signal was delivered
- Stopping all the threads in the process
- · Resuming the execution of all the threads in the process

An application program can use the *sigaction* system call to specify an action for a signal, including these choices:

Table 4.5 Signals defined in FreeBSD.

Name	Default action	Description
SIGHUP	terminate process	terminal line hangup
SIGINT	terminate process	interrupt program
SIGQUIT	create core image	quit program
SIGILL	create core image	illegal instruction
SIGTRAP	create core image	trace trap
SIGABRT	create core image	abort
SIGEMT	create core image	emulate instruction executed
SIGFPE	create core image	floating-point exception
SIGKILL	terminate process	kill program
SIGBUS	create core image	bus error
SIGSEGV	create core image	segmentation violation
SIGSYS	create core image	bad argument to system call
SIGPIPE	terminate process	write on a pipe with no one to read it
SIGALRM	terminate process	real-time timer expired
SIGTERM	terminate process	software termination signal
SIGURG	discard signal	urgent condition on I/O channel
SIGSTOP	stop process	stop signal not from terminal
SIGTSTP	stop process	stop signal from terminal
SIGCONT	discard signal	a stopped process is being continued
SIGCHLD	discard signal	notification to parent on child stop or exit
SIGTTIN	stop process	read on terminal by background process
SIGTTOU	stop process	write to terminal by background process
SIGIO	discard signal	I/O possible on a descriptor
SIGXCPU	terminate process	CPU time limit exceeded
SIGXFSZ	terminate process	file-size limit exceeded
SIGVTALRM	terminate process	virtual timer expired
SIGPROF	terminate process	profiling timer expired
SIGWINCH	discard signal	window size changed
SIGINFO	discard signal	information request
SIGUSR1	terminate process	user-defined signal 1
SIGUSR2	terminate process	user-defined signal 2
SIGTHR	terminate process	used by thread library
SIGLIBRT	terminate process	used by real-time library

- Taking the default action
- Ignoring the signal
- Catching the signal with a *handler*

A *signal handler* is a user-mode routine that the system will invoke when the signal is received by the process. The handler is said to catch the signal. The two signals SIGSTOP and SIGKILL cannot be masked, ignored, or caught; this restriction ensures that a software mechanism exists for stopping and killing runaway processes. It is not possible for a process to decide which signals would cause the creation of a core file by default, but it is possible for a process to prevent the creation of such a file by ignoring, blocking, or catching the signal.

Signals are posted to a process by the system when it detects a hardware event, such as an illegal instruction, or a software event, such as a stop request from the terminal. A signal may also be posted by another process through the *kill* system call. A sending process may post signals to only those receiving processes that have the same effective user identifier (unless the sender is the superuser). A single exception to this rule is the *continue signal*, SIGCONT, which always can be sent to any descendant of the sending process. The reason for this exception is to allow users to restart a *setuid* program that they have stopped from their keyboard.

Like hardware interrupts, each thread in a process can mask the delivery of signals. The execution state of each thread contains a set of signals currently masked from delivery. If a signal posted to a thread is being masked, the signal is recorded in the thread's set of pending signals, but no action is taken until the signal is unmasked. The *sigprocmask* system call modifies the set of masked signals for a thread. It can add to the set of masked signals, delete from the set of masked signals, or replace the set of masked signals. Although the delivery of the SIGCONT signal to the signal handler of a process may be masked, the action of resuming that stopped process is not masked.

Two other signal-related system calls are *sigsuspend* and *sigaltstack*. The *sigsuspend* call permits a thread to relinquish the processor until that thread receives a signal. This facility is similar to the system's *sleep()* routine. The *sigaltstack* call allows a process to specify a run-time stack to use in signal delivery. By default, the system will deliver signals to a process on the latter's normal run-time stack. In some applications, however, this default is unacceptable. For example, if an application has many threads that have carved up the normal run-time stack into many small pieces, it is far more memory efficient to create one large signal stack on which all the threads handle their signals than it is to reserve space for signals on each thread's stack.

The final signal-related facility is the *sigreturn* system call. *Sigreturn* is the equivalent of a user-level load-processor-context operation. The kernel is passed a pointer to a (machine-dependent) context block that describes the user-level execution state of a thread. The *sigreturn* system call restores state and resumes execution after a normal return from a user's signal handler.

Posting of a Signal

The implementation of signals is broken up into two parts: posting a signal to a process and recognizing the signal and delivering it to the target thread. Signals may be posted by any process or by code that executes at interrupt level. Signal

delivery normally takes place within the context of the receiving thread. When a signal forces a process to be stopped, the action can be carried out on all the threads associated with that process when the signal is posted.

A signal is posted to a single process with the *psignal()* routine or to a group of processes with the *gsignal()* routine. The *gsignal()* routine invokes *psignal()* for each process in the specified process group. The actions associated with posting a signal are straightforward, but the details are messy. In theory, posting a signal to a process simply causes the appropriate signal to be added to the set of pending signals for the appropriate thread within the process, and the selected thread is then set to run (or is awakened if it was sleeping at an interruptible priority level).

The disposition of signals is set on a per-process basis. The kernel first checks to see if the signal should be ignored, in which case it is discarded. If the process has specified the default action, then the default action is taken. If the process has specified a signal handler that should be run, then the kernel must select the appropriate thread within the process that should handle the signal. When a signal is raised because of the action of the currently running thread (for example, a segment fault), the kernel will only try to deliver it to that thread. If the thread is masking the signal, then the signal will be held pending until it is unmasked. When a process-related signal is sent (for example, an interrupt), then the kernel searches all the threads associated with the process, searching for one that does not have the signal masked. The signal is delivered to the first thread that is found with the signal unmasked. If all threads associated with the process are masking the signal, then the signal is left in the list of signals pending for the process for later delivery.

Each time that a thread returns from a call to sleep() (with the PCATCH flag set) or prepares to exit the system after processing a system call or trap, it uses the cursig() routine to check whether a signal is pending delivery. The cursig() routine determines the next signal that should be delivered to a thread by inspecting the process's signal list, $p_siglist$, to see if it has any signals that should be propagated to the thread's signal list, $td_siglist$. It then inspects the $td_siglist$ field to check for any signals that should be delivered to the thread. If a signal is pending and must be delivered in the thread's context, it is removed from the pending set, and the thread invokes the postsig() routine to take the appropriate action.

The work of *psignal()* is a patchwork of special cases required by the process-debugging and job-control facilities and by intrinsic properties associated with signals. The steps involved in posting a signal are as follows:

 Determine the action that the receiving process will take when the signal is delivered. This information is kept in the *p_sigignore* and *p_sigcatch* fields of the process's process structure. If a process is not ignoring or catching a signal, the default action is presumed to apply. If a process is being traced by its parent—that is, by a debugger—the parent process is always permitted to intercede before the signal is delivered. If the process is ignoring the signal, *psignal()*'s work is done and the routine can return.

- 2. Given an action, *psignal()* selects the appropriate thread and adds the signal to the thread's set of pending signals, *td_siglist*, and then does any implicit actions specific to that signal. For example, if the signal is the continue signal, SIGCONT, any pending signals that would normally cause the process to stop, such as SIGTTOU, are removed.
- 3. Next, *psignal()* checks whether the signal is being masked. If the thread is currently masking delivery of the signal, *psignal()*'s work is complete and it may return.
- 4. If the signal is not being masked, *psignal()* must either perform the action directly or arrange for the thread to execute so that the thread will take the action associated with the signal. Before setting the thread to a runnable state, *psignal()* must take different courses of action depending on the state of the thread as follows:
- SLEEPING The thread is blocked awaiting an event. If the thread is sleeping noninterruptibly, then nothing further can be done. Otherwise, the kernel can apply the action—either directly or indirectly—by waking up the thread. There are two actions that can be applied directly. For signals that cause a process to stop, all the threads in the process are placed in the STOPPED state, and the parent process is notified of the state change by a SIGCHLD signal being posted to it. For signals that are ignored by default, the signal is removed from the signal list and the work is complete. Otherwise, the action associated with the signal must be done in the context of the receiving thread, and the thread is placed onto the run queue with a call to *setrunnable()*.
- STOPPED The process is stopped by a signal or because it is being debugged. If the process is being debugged, then there is nothing to do until the controlling process permits it to run again. If the process is stopped by a signal and the posted signal would cause the process to stop again, then there is nothing to do, and the posted signal is discarded. Otherwise, the signal is either a continue signal or a signal that would normally cause the process to terminate (unless the signal is caught). If the signal is SIGCONT, then all the threads in the process that were previously running are set running again. Any threads in the process that were blocked waiting on an event are returned to the SLEEPING state. If the signal is SIGKILL, then all the threads in the process are set running again no matter what, so that they can terminate the next time that they are scheduled to run. Otherwise, the signal causes the threads in the process to be made runnable, but the threads are not placed on the run queue because they must wait for a continue signal.

RUNNABLE, NEW, ZOMBIE

If a thread scheduled to receive a signal is not the currently executing thread, its TDF_NEEDRESCHED flag is set, so that the signal will be noticed by the receiving thread as soon as possible.



Figure 4.6 Delivery of a signal to a process. Step 1: The kernel places a signal context on the user's stack. Step 2: The kernel places a signal-handler frame on the user's stack and arranges to start running the user process in the *sigtramp()* code. When the *sigtramp()* routine starts running, it calls the user's signal handler. Step 3: The user's signal handler returns to the *sigtramp()* routine, which pops the signal-handler context from the user's stack. Step 4: The *sigtramp()* routine finishes by calling the *sigreturn* system call, which restores the previous user context from the signal context, pops the signal context from the stack, and resumes the user's process at the point at which it was running before the signal occurred.

Delivering a Signal

Most actions associated with delivering a signal to a thread are carried out within the context of that thread. A thread checks its $td_siglist$ field for pending signals at least once each time that it enters the system by calling cursig().

If cursig() determines that there are any unmasked signals in the thread's signal list, it calls issignal() to find the first unmasked signal in the list. If delivering the signal causes a signal handler to be invoked or a core dump to be made, the caller is notified that a signal is pending, and the delivery is done by a call to postsig(). That is,

```
if (sig = cursig(curthread))
    postsig(sig);
```

Otherwise, the action associated with the signal is done within *issignal()* (these actions mimic the actions carried out by *psignal()*).

The *postsig()* routine has two cases to handle:

- 1. Producing a core dump
- 2. Invoking a signal handler

The former task is done by the *coredump()* routine and is always followed by a call to *exit()* to force process termination. To invoke a signal handler, *postsig()* first calculates a set of masked signals and installs that set in $td_sigmask$. This set normally includes the signal being delivered, so that the signal handler will not be invoked recursively by the same signal. Any signals specified in the *sigaction* system call at the time the handler was installed also will be included. The *postsig()* routine then calls the *sendsig()* routine to arrange for the signal handler to execute immediately after the thread returns to user mode. Finally, the signal in $td_siglist$ is cleared and *postsig()* returns, presumably to be followed by a return to user mode.

The implementation of the sendsig() routine is machine dependent. Figure 4.6 shows the flow of control associated with signal delivery. If an alternate stack has been requested, the user's stack pointer is switched to point at that stack. An argument list and the thread's current user-mode execution context are stored by the kernel on the (possibly new) stack. The state of the thread is manipulated so that, on return to user mode, a call will be made immediately to a body of code termed the signal-trampoline code. This code invokes the signal handler (between steps 2 and 3 in Figure 4.6) with the appropriate argument list, and, if the handler returns, makes a *sigreturn* system call to reset the thread's signal state to the state that existed before the signal. The signal-trampoline code, sigcode() contains several assembly-language instructions that are copied onto the thread's stack when the signal is about to be delivered. It is the responsibility of the trampoline code to call the registered signal handler, handle any possible errors, and then return the thread to normal execution. The trampoline code is implemented in assembly language because it must directly manipulate CPU registers, including those relating to the stack and return value.

4.8 Process Groups and Sessions

Each process in the system is associated with a *process group*. The group of processes in a process group is sometimes referred to as a *job* and is manipulated as a single entity by processes such as the shell. Some signals (e.g., SIGINT) are delivered to all members of a process group, causing the group as a whole to suspend or resume execution, or to be interrupted or terminated.

Sessions were designed by the IEEE POSIX.1003.1 Working Group with the intent of fixing a long-standing security problem in UNIX—namely, that processes could modify the state of terminals that were trusted by another user's processes. A *session* is a collection of process groups, and all members of a process group are members of the same session. In FreeBSD, when a user first logs onto the system, he is entered into a new session. Each session has a *controlling process*,

which is normally the user's login shell. All subsequent processes created by the user are part of process groups within this session, unless he explicitly creates a new session. Each session also has an associated login name, which is usually the user's login name. This name can be changed by only the superuser.

Each session is associated with a terminal, known as its *controlling terminal*. Each controlling terminal has a process group associated with it. Normally, only processes that are in the terminal's current process group read from or write to the terminal, allowing arbitration of a terminal between several different jobs. When the controlling process exits, access to the terminal is taken away from any remaining processes within the session.

Newly created processes are assigned process IDs distinct from all alreadyexisting processes and process groups, and are placed in the same process group and session as their parent. Any process may set its process group equal to its process ID (thus creating a new process group) or to the value of any process group within its session. In addition, any process may create a new session, as long as it is not already a process-group leader.

Process Groups

A process group is a collection of related processes, such as a shell pipeline, all of which have been assigned the same *process-group identifier*. The process-group identifier is the same as the PID of the process group's initial member; thus, process-group identifiers share the namespace of process identifiers. When a new process group is created, the kernel allocates a process-group structure to be associated with it. This process-group structure is entered into a process-group hash table so that it can be found quickly.

A process is always a member of a single process group. When it is created, each process is placed into the process group of its parent process. Programs such as shells create new process groups, usually placing related child processes into a group. A process can change its own process group or that of one of its child process by creating a new process group or by moving a process into an existing process group using the *setpgid* system call. For example, when a shell wants to set up a new pipeline, it wants to put the processes in the pipeline into a process group different from its own so that the pipeline can be controlled independently of the shell. The shell starts by creating the first process in the pipeline, which initially has the same process-group identifier as the shell. Before executing the target program, the first process does a *setpgid* to set its process group, with the child process as the *process-group leader* of the process group. As the shell starts each additional process for the pipeline, each child process uses *setpgid* to join the existing process group.

In our example of a shell creating a new pipeline, there is a *race condition*. As the additional processes in the pipeline are spawned by the shell, each is placed in the process group created by the first process in the pipeline. These conventions are enforced by the *setpgid* system call. It restricts the set of process-group identifiers to which a process may be set to either a value equal to its own PID or

to a value of another process-group identifier in its session. Unfortunately, if a pipeline process other than the process-group leader is created before the processgroup leader has completed its *setpgid* call, the *setpgid* call to join the process group will fail. As the setpgid call permits parents to set the process group of their children (within some limits imposed by security concerns), the shell can avoid this race by making the *setpgid* call to change the child's process group both in the newly created child and in the parent shell. This algorithm guarantees that, no matter which process runs first, the process group will exist with the correct process-group leader. The shell can also avoid the race by using the vfork variant of the *fork* system call that forces the parent process to wait until the child process either has done an exec system call or has exited. In addition, if the initial members of the process group exit before all the pipeline members have joined the group-for example, if the process-group leader exits before the second process joins the group, the setpgid call could fail. The shell can avoid this race by ensuring that all child processes are placed into the process group without calling the wait system call, usually by blocking the SIGCHLD signal so that the shell will not be notified of a child exit until after all the children have been placed into the process group. As long as a process-group member exists, even as a zombie process, additional processes can join the process group.

There are additional restrictions on the *setpgid* system call. A process may join process groups only within its current session (discussed in the next section), and it cannot have done an *exec* system call. The latter restriction is intended to avoid unexpected behavior if a process is moved into a different process group after it has begun execution. Therefore, when a shell calls *setpgid* in both parent and child processes after a *fork*, the call made by the parent will fail if the child has already made an *exec* call. However, the child will already have joined the process group successfully, and the failure is innocuous.

Sessions

Just as a set of related processes are collected into a process group, a set of process groups are collected into a *session*. A session is a set of one or more process groups and may be associated with a terminal device. The main uses for sessions are to collect a user's login shell and the jobs that it spawns and to create an isolated environment for a daemon process and its children. Any process that is not already a process-group leader may create a session using the *setsid* system call, becoming the *session leader* and the only member of the session. Creating a session also creates a new process group, where the process-group leader. By definition, all members of a process group are members of the same session.

A session may have an associated *controlling terminal* that is used by default for communicating with the user. Only the session leader may allocate a controlling terminal for the session, becoming a *controlling process* when it does so. A device can be the controlling terminal for only one session at a time. The terminal I/O system (described in Section 8.6) synchronizes access to a terminal by permitting only a single process group to be the foreground process group for a controlling terminal at any time. Some terminal operations are restricted to members of the session. A session can have at most one controlling terminal. When a session is created, the session leader is dissociated from its controlling terminal if it had one.

A login session is created by a program that prepares a terminal for a user to log into the system. That process normally executes a shell for the user, and thus the shell is created as the controlling process. An example of a typical login session is shown in Figure 4.7.

The data structures used to support sessions and process groups in FreeBSD are shown in Figure 4.8. This figure parallels the process layout shown in Figure 4.7. The $pg_members$ field of a process-group structure heads the list of member processes; these processes are linked together through the p_pglist list entry in the process structure. In addition, each process has a reference to its process-group structure has a pointer to its enclosing session. The session structure tracks per-login information, including the process that created and controls the session, the controlling terminal for the session, and the login name associated with the session. Two processes wanting to determine whether they are in the same session can traverse their p_pgrp pointers to find their process-group structures and then compare the $pg_session$ pointers to see whether the latter are the same.

Job Control

Job control is a facility first provided by the C shell [Joy, 1994] and today is provided by most shells. It permits a user to control the operation of groups of processes termed **jobs**. The most important facilities provided by job control are the abilities to suspend and restart jobs and to do the multiplexing of access to the

Figure 4.7 A session and its processes. In this example, process 3 is the initial member of the session—the session leader—and is referred to as the controlling process if it has a controlling terminal. It is contained in its own process group, 3. Process 3 has spawned two jobs: One is a pipeline composed of processes 4 and 5, grouped together in process group 4, and the other one is process 8, which is in its own process group, 8. No process-group leader can create a new session; thus, process 3, 4, or 8 could not start its own session, but process 5 would be allowed to do so.





Figure 4.8 Process-group organization.

user's terminal. Only one job at a time is given control of the terminal and is able to read from and write to the terminal. This facility provides some of the advantages of window systems, although job control is sufficiently different that it is often used in combination with window systems. Job control is implemented on top of the process group, session, and signal facilities.

Each job is a process group. Outside the kernel, a shell manipulates a job by sending signals to the job's process group with the *killpg* system call, which delivers a signal to all the processes in a process group. Within the system, the two main users of process groups are the terminal handler (Section 8.6) and the interprocess-communication facilities (Chapter 12). Both facilities record process-group identifiers in private data structures and use them in delivering signals. The terminal handler, in addition, uses process groups to multiplex access to the controlling terminal.

For example, special characters typed at the keyboard of the terminal (e.g., control-C or control-) result in a signal being sent to all processes in one job in

the session; that job is in the *foreground*, whereas all other jobs in the session are in the *background*. A shell may change the foreground job by using the *tcsetpgrp()* function, implemented by the TIOCSPGRP *ioctl* on the controlling terminal. Background jobs will be sent the SIGTTIN signal if they attempt to read from the terminal, normally stopping the job. The SIGTTOU signal is sent to background jobs that attempt an *ioctl* system call that would alter the state of the terminal. The SIGTTOU signal is also sent if the TOSTOP option is set for the terminal, and an attempt is made to write to the terminal.

The foreground process group for a session is stored in the t_pgrp field of the session's controlling terminal *tty* structure (see Section 8.6). All other process groups within the session are in the background. In Figure 4.8, the session leader has set the foreground process group for its controlling terminal to be its own process group. Thus, its two jobs are in the background, and the terminal input and output will be controlled by the session-leader shell. Job control is limited to processes contained within the same session are permitted to reassign the controlling terminal among the process groups within the session.

If a controlling process exits, the system revokes further access to the controlling terminal and sends a SIGHUP signal to the foreground process group. If a process such as a job-control shell exits, each process group that it created will become an orphaned process group: a process group in which no member has a parent that is a member of the same session but of a different process group. Such a parent would normally be a job-control shell capable of resuming stopped child processes. The pg jobc field in Figure 4.8 counts the number of processes within the process group that have the controlling process as a parent. When that count goes to zero, the process group is orphaned. If no action were taken by the system, any orphaned process groups that were stopped at the time that they became orphaned would be unlikely ever to resume. Historically, the system dealt harshly with such stopped processes: They were killed. In POSIX and FreeBSD, an orphaned process group is sent a hangup and a continue signal if any of its members are stopped when it becomes orphaned by the exit of a parent process. If processes choose to catch or ignore the hangup signal, they can continue running after becoming orphaned. The system keeps a count of processes in each process group that have a parent process in another process group of the same session. When a process exits, this count is adjusted for the process groups of all child processes. If the count reaches zero, the process group has become orphaned. Note that a process can be a member of an orphaned process group even if its original parent process is still alive. For example, if a shell starts a job as a single process A, that process then forks to create process B, and the parent shell exits; then process B is a member of an orphaned process group but is not an orphaned process.

To avoid stopping members of orphaned process groups if they try to read or write to their controlling terminal, the kernel does not send them SIGTTIN and SIGTTOU signals, and prevents them from stopping in response to those signals. Instead, their attempts to read or write to the terminal produce an error.

4.9 Process Debugging

FreeBSD provides a simple facility for controlling and debugging the execution of a process. This facility, accessed through the *ptrace* system call, permits a parent process to control a child process's execution by manipulating user- and kernel-mode execution states. In particular, with *ptrace*, a parent process can do the following operations on a child process:

- Attaches to an existing process to begin debugging it
- · Reads and writes address space and registers
- · Intercepts signals posted to the process
- Single steps and continues the execution of the process
- Terminates the execution of the process

The *ptrace* call is used almost exclusively by program debuggers, such as **lldb**.

When a process is being traced, any signals posted to that process cause it to enter the STOPPED state. The parent process is notified with a SIGCHLD signal and may interrogate the status of the child with the *wait4* system call. On most machines, *trace traps*, generated when a process is single stepped, and *breakpoint faults*, caused by a process executing a breakpoint instruction, are translated by FreeBSD into SIGTRAP signals. Because signals posted to a traced process cause it to stop and result in the parent being notified, a program's execution can be controlled easily.

To start a program that is to be debugged, the debugger first creates a child process with a *fork* system call. After the fork, the child process uses a *ptrace* call that causes the process to be flagged as "traced" by setting the P_TRACED bit in the *p_flag* field of the process structure. The child process then sets the trace trap bit in the process's processor status word and calls *execve* to load the image of the program that is to be debugged. Setting this bit ensures that the first instruction executed by the child process after the new image is loaded will result in a hardware trace trap, which is translated by the system into a SIGTRAP signal. Because the parent process is notified about all signals to the child, it can intercept the signal and gain control over the program before it executes a single instruction.

Alternatively, the debugger may take over an existing process by attaching to it. A successful attach request causes the process to enter the STOPPED state and to have its P_TRACED bit set in the p_flag field of its process structure. The debugger can then begin operating on the process in the same way as it would with a process that it had explicitly started.

An alternative to the *ptrace* system call is the **/proc** filesystem. The functionality provided by the **/proc** filesystem is the same as that provided by *ptrace*; it differs only in its interface. The **/proc** filesystem implements a view of the system process table inside the filesystem and is so named because it is normally mounted on **/proc**. It provides a two-level view of process space. At the highest level, processes themselves are named, according to their process IDs. There is also a special node called **curproc** that always refers to the process making the lookup request.

Each node is a directory that contains the following entries:

- **ctl** A write-only file that supports a variety of control operations. Control commands are written as strings to the **ctl** file. The control commands are:
 - **attach** Stops the target process and arranges for the sending process to become the debug control process.
 - **detach** Continues execution of the target process and remove it from control by the debug process (that need not be the sending process).
 - **run** Continues running the target process until a signal is delivered, a breakpoint is hit, or the target process exits.
 - **step** Single steps the target process, with no signal delivery.
 - wait Waits for the target process to come to a steady state ready for debugging. The target process must be in this state before any of the other commands are allowed.

The string can also be the name of a signal, lowercase and without the SIG prefix, in which case that signal is delivered to the process.

- dbregs Sets the debug registers as defined by the machine architecture.
- etype The type of the executable referenced by the file entry.
- **file** A reference to the vnode from which the process text was read. This entry can be used to gain access to the symbol table for the process or to start another copy of the process.
- **fpregs** The floating point registers as defined by the machine architecture. It is only implemented on machines that have distinct general-purpose and floating-point register sets.
- **map** A map of the process's virtual memory.
- **mem** The complete virtual memory image of the process. Only those addresses that exist in the process can be accessed. Reads and writes to this file modify the process. Writes to the text segment remain private to the process. Because the address space of another process can be accessed with *read* and *write* system calls, a debugger can access a process being debugged with much greater efficiency than it can with the *ptrace* system call. The pages of interest in the process being debugged are mapped into the kernel address space. The data requested by the debugger can then be copied directly from the kernel to the debugger's address space.

- regs Allows read and write access to the register set of the process.
- rlimit A read-only file containing the process's current and maximum limits.
- **status** The process status. This file is read-only and returns a single line containing multiple space-separated fields that include the command name, the process id, the parent process id, the process group id, the session id, the controlling terminal (if any), a list of the process flags, the process start time, user and system times, the wait channel message, and the process credentials.

Each node is owned by the process's user and belongs to that user's primary group, except for the **mem** node, which belongs to the *kmem* group.

In a normal debugging environment, where the target does a *fork* followed by an *exec* by the debugger, the debugger should *fork* and the child should stop itself (with a self-inflicted SIGSTOP, for example). The parent should issue a *wait* and then an *attach* command via the appropriate **ctl** file. The child process will receive a SIGTRAP immediately after the call to *exec*.

Users wishing to view process information often find it easier to use the **proc**stat command than to figure out how to extract the information from the /**proc** filesystem.

Exercises

- 4.1 For each state listed in Table 4.1, list the system queues on which a process in that state might be found.
- 4.2 Why is the performance of the context-switching mechanism critical to the performance of a highly multiprogrammed system?
- 4.3 What effect would increasing the time quantum have on the system's interactive response and total throughput?
- 4.4 What effect would reducing the number of run queues from 64 to 32 have on the scheduling overhead and on system performance?
- 4.5 Give three reasons for the system to select a new process to run.
- 4.6 Describe the three types of scheduling policies provided by FreeBSD.
- 4.7 What type of jobs does the timeshare scheduling policy favor? Propose an algorithm for identifying these favored jobs.
- 4.8 When and how does thread scheduling interact with memory-management facilities?
- 4.9 After a process has exited, it may enter the state of being a ZOMBIE before disappearing from the system entirely. What is the purpose of the ZOMBIE state? What event causes a process to exit from ZOMBIE?

- 4.10 Suppose that the data structures shown in Table 4.3 do not exist. Instead, assume that each process entry has only its own PID and the PID of its parent. Compare the costs in space and time to support each of the following operations:
 - a. Creation of a new process
 - b. Lookup of the process's parent
 - c. Lookup of all of a process's siblings
 - d. Lookup of all of a process's descendants
 - e. Destruction of a process
- 4.11 What are the differences between a mutex and a lock-manager lock?
- 4.12 Give an example of where a mutex lock should be used. Give an example of where a lock-manager lock should be used.
- 4.13 A process blocked without setting the PCATCH flag may never be awakened by a signal. Describe two problems a noninterruptible sleep may cause if a disk becomes unavailable while the system is running.
- 4.14 Describe the limitations a jail puts on the filesystem namespace, network access, and processes running in the jail.
- *4.15 In FreeBSD, the signal SIGTSTP is delivered to a process when a user types a "suspend character." Why would a process want to catch this signal before it is stopped?
- *4.16 Before the FreeBSD signal mechanism was added, signal handlers to catch the SIGTSTP signal were written as

```
catchstop()
{
    prepare to stop;
    signal(SIGTSTP, SIG_DFL);
    kill(getpid(), SIGTSTP);
    signal(SIGTSTP, catchstop);
}
```

This code causes an infinite loop in FreeBSD. Why does it do so? How should the code be rewritten?

- *4.17 The process-priority calculations and accounting statistics are all based on sampled data. Describe hardware support that would permit more accurate statistics and priority calculations.
- *4.18 Why are signals a poor interprocess-communication facility?

- **4.19 A kernel-stack-invalid trap occurs when an invalid value for the kernelmode stack pointer is detected by the hardware. How might the system gracefully terminate a process that receives such a trap while executing on its kernel-run-time stack?
- **4.20 Describe alternatives to the test-and-set instruction that would allow you to build a synchronization mechanism for a multiprocessor FreeBSD system.
- **4.21 A lightweight process is a thread of execution that operates within the context of a normal FreeBSD process. Multiple lightweight processes may exist in a single FreeBSD process and share memory, but each is able to do blocking operations, such as system calls. Describe how lightweight processes might be implemented entirely in user mode.

References

Aral et al., 1989.

Z. Aral, J. Bloom, T. Doeppner, I. Gertner, A. Langerman, & G. Schaffer, "Variable Weight Processes with Flexible Shared Resources," *USENIX Association Conference Proceedings*, pp. 405–412, January 1989.

Dekker, 2013.

Dekker, "Dekker Algorithm," *Wikipedia*, available from http: //en.wikipedia.org/wiki/Dekkers_algorithm, November 2013.

Joy, 1994.

W. N. Joy, "An Introduction to the C Shell," in *4.4BSD User's Supplementary Documents*, pp. 4:1–46, O'Reilly & Associates, Inc., Sebastopol, CA, 1994. McDougall & Mauro, 2006.

R. McDougall & J. Mauro, *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture (2nd Edition)*, Prentice Hall, Upper Saddle River, NJ, 2006.

Ritchie, 1988.

D. M. Ritchie, "Multi-Processor UNIX," private communication, April 25, 1988.

Roberson, 2003.

J. Roberson, "ULE: A Modern Scheduler For FreeBSD," *Proceedings of the USENIX BSDCon 2003*, pp. 17–28, September 2003.

Simpleton, 2008.

Caffeinated Simpleton, *A Threading Model Overview*, available from http://justin.harmonize.fm / Development / 2008 / 09 / 09 / threading-model-overview.html, September 2008.

/, 45–46 ., 443, 450, 474–475, 814 .., 443, 450–451, 462, 474–475, 478, 814 #!, 70 .sujournal, 488

A

ABI. See application binary interface absolute pathname, 46, 807, 828 accept system call, 597-598, 611-614, 646, 664.739 definition, 597 access control, 29-34, 47-48, 150-174, 184-200, 803-805 commands, 162, 169-172 functions, NFS version 4, 173-174 interprocess, 34, 159-161, 182 list, 30-32, 48, 150, 154, 162-174, 436-437, 525, 573-574, 807, 814 list, default, 170 NFS, 573 access rights, 608, 617 receiving, 620 access system call, 353 access vnode operator, 432 access_mask, 172-173 accounting, process resource, 31, 67, 129, 790,800

accton, 799 ACL. See access control list acl structure, 166 acl_denies(), 172-173 acl entry structure, 166 ACPI. See advanced configuration and power interface active page list, 290 adaptive idle, 125 adaptive replacement cache, 525, 539, 547-548 address family, 596, 608, 611, 807 address resolution protocol, 641, 655-657, 669,807 implementation of, 655-657 purpose of, 655 address-space management, process, 228-230 address space. See virtual address space address structure Internet, 612 local domain, 612 socket, 182, 596, 611, 839 address translation, 222, 807 addresses, IPv6, 660-662 adjtime system call, 74 advanced configuration and power interface, 53, 363-364, 777, 781, 783 advanced-encryption standard, 210, 213 block cipher, 213, 215

Advanced Micro Devices Corporation, 25, 285, 362, 420-421, 784 virtualization, 421-422 advanced programmable interrupt controller, 363-364, 420, 423, 790 advanced-technology-attachment disk. 363-364, 399, 402, 409-410 advisory locking, 323, 432, 807 advlock vnode operator, 432 AES. See advanced-encryption standard AFS. See Andrew filesystem AH. See authentication header Ahrens, Matt, xxvi aio error system call, 321-322, 330 aio_read system call, 321, 359 aio_return system call, 322, 330 aio_suspend system call, 322, 330 aio_waitcomplete system call, 322 aio write system call, 321, 359 algorithm for disksort(), 376 elevator sorting, 375, 418 mbuf storage-management, 605 for physical I/O, 372 TCP. 732-741 TCP slow-start, 752-756 Allman, Eric, xxix allocation descriptor, 614 directory space, 444-445 extent-based, 516-517 FFS file block, 507, 511-513, 809 FFS fragment, 512-514 inode, 434 kernel address space, 233-244, 787 kernel memory, 38-39 kernel resource, 259-260 mbuf, 605, 795 process identifier, 127 virtual memory map, 304-305 ZFS file block, 542-543 allocator dynamic per-CPU, 789 keg, 238 slab, 236-237, 787, 791 virtual-network stack, 789 zone, 239-241, 791, 793 allocbuf(), 351 allocdirect structure, 466-468, 470-471

allocindir structure, 467, 470-471 ambient authority, 151, 174-180, 807 AMD. See Advanced Micro Devices Corporation AMD-V. See Advanced Micro Devices Corporation virtualization ancillary data, 598, 616-618, 808 Andrew filesystem, 552 anonymous object, 245, 248, 808 AOUT executable format, 70 API. See application programming interface APIC. See advanced programmable interrupt controller append-only file, 439 Apple OS/X operating system, 3, 436, 445 application binary interface, 403, 783 application, client-server, 50 application compartmentalization, 30, 149, 151, 174-175, 808 application programming interface, 34, 51, 114, 166, 187, 191, 198, 201, 403, 700 ARC. See adaptive replacement cache arc4random(), 209 architecture ARM, 405, 782, 784 MIPS, 7, 405, 782-784, 790 PC, 362-364 PPC, 405 SPARC64, 790 arguments, marshalling of, 553, 824 ARM architecture, 405, 782, 784 ARP. See address resolution protocol ARPANET, 6, 650 assembly-language startup, 783-784 assembly language in the kernel, 25, 61, 116, 370, 782-784, 786 association, Internet, 721-723 association setup, SCTP, 761-764 association shutdown, SCTP, 766 assured pipeline, 176 AST. See asynchronous system trap asymmetric cryptography, 206, 700, 808 asynchronous I/O, 320-322, 326 interrupt, 58, 60, 99 logging facility, 84 system trap, 116-117, 808 transfer mode, 707-708 ATA. See advanced-technology-attachment disk

ATM. See asynchronous transfer mode AT&T, xxi, xxii, 6-9, 11-12 ATTACHED flag, 466 definition, 466 attribute manipulation, filesystem, 432 attribute update, filestore, 497 attributes, extended, 436-438 attributes, system extended, 168 audit alarm entry, 171 allow entry, 171 deny entry, 171 event, 35, 200-205 informational entry, 171 pipe, 200-205 preselection, 201, 204 queue, 204-205 record, 35, 200-205 system call, 201, 203, 205 trail, 35, 200-205 UID, 35, 201-202, 204 worker thread, 204-205 auditing, security-event, 30-31, 35, 149, 151, 200-205, 792-793, 800, 836 audit_init(), 793 auditreduce, 35 AUID. See audit UID authentication, 30, 35 data, 691, 693 header, 663-664, 689-691, 693, 697 autoconfiguration, 402, 660, 808 4.4BSD, 403 contribution of, 8 data structures, 407-410 device driver support for, 369, 403-413 IPv6, 666-670 phase, 404 resource, 412-413

B

```
B programming language, 4
back ends, device driver, 414–428
background fsck, 486
background process, 141, 387, 808, 817
backing storage, 221, 397, 809
bare-metal system library, 780
basic input-output system, 52–53, 364, 377,
775–779, 781, 790
```

basic kernel services, 787–792 basic security module, 35, 201-203, 205 bawrite(), 348 BCPL programming language, 4 bdwrite(), 348 Bell Laboratories, 3-5 benefit of global vnode table, 345 Berkeley packet filter, 700-701, 703 macro hook, 701 Berkeley Software Design Inc., 11-13 best fit, 234, 512-513 bhyve, 184, 414-415 Biba integrity policy, 151, 158, 186–187, 189-190, 195, 200, 797 bind system call, 182, 664, 723, 767 definition, 597 biodone(), 402, 469 BIOS. See basic input-output system biowait(), 402bitmap dependencies, soft updates, 466-467 black-hole route, 680, 809 blkatoff vnode operator, 497-498 block, 433, 809 clustering, 498, 505-507, 514-517, 811 I/O, 375, 498-501, 505, 543 interface, Xen, 427 protection, GELI, 215-216 size, 368, 502, 809 bmsafemap structure, 466-467, 470, 472, 490 boot, 775-801, 817 boot blocks, FFS, 503-504 boot, cryptographically verified, 777 boot device, 776-777, 783 /boot/device.hints, 404 boot flags, 779, 781, 783 /boot/kernel/kernel, 781 boot loader, 777-789 1st-stage, 777-779 2nd-stage, 779 final-stage, 779-781 /boot/loader, 779-782 /boot/loader.4th, 781 /boot/loader.conf, 781 boot menu, 778-781 boot partition, FreeBSD, 779 boot-time diagnostics, 776 boot2, 782 /boot.config, 779

bootinfo structure, 781, 783 bootstrapping, 25, 52, 809 setting time when, 73 see also boot bottom half of device driver, 369 kernel, 59-60, 809 terminal driver, 384 BPF. See Berkeley packet filter bgrelse(), 349 bread(), 348, 350-351, 375 break character, 390 breakpoint fault, 142, 809 brelse(), 348bremfree(), 350 broadcast message, 636, 674, 725, 809 address, 636, 653-654 IP handling of, 672 BSD, obtaining, xxvi BSD, open source, 9-14 BSDI. See Berkeley Software Design Inc. BSM. See basic security module bss segment, 69, 263, 784, 809 BTX. See i386 boot extender buf structure, 375 bufdaemon, 58 buffer cache, 347, 435, 499-501, 790, 793-794,809 4.4BSD, 302 consistency, 351 effectiveness, 347 implementation of, 350-351 interface, 348-349 management, 347-351 memory allocation, 351 structure of, 349-350 buffer list CLEAN, 350, 359 DIRTY, 350, 359 **EMPTY**, 350 LOCKED, 349 buffer update, 468-470 buffer wait, 468-470, 473-474, 476, 496 buffering filesystem, 499-501 network, 643-644 policy, protocol, 643 bufinit(), 790 bus_add_child(), 413

bus_child_detached(),413
bus_driver_added(),413
bus_probe_nomatch(),413
bus_read_ivar(),413
bus_write_ivar(),413
bwrite(),348,375

С

C-language startup, 784-785 C library, 73 system calls in the, 62 C programming language, 3-4, 26, 62 cache alias, virtual-memory, 282 directory offset, 446-447 filename, 346-347, 795 inode, 442-443 page list, 290 vnode, 249 caching delegation and callbacks, 574-581 calendar queue, 121 call gates, 150 callback, 567, 579, 610, 810 callout queue, 67-69, 733, 795, 810 callout callwheel init(), 790 CAM. See common access method camisr(), 402camisr_runqueue(), 402 canonical mode, 383, 810 capability, 176-180, 810 discipline, 179 mode, 31-32, 176, 179-180 refinement, 176 system, 174-180, 810 system model, 149, 176 $cap_check(), 179$ *cap* enter system call, 179 cap_getmode system call, 179 CAPP. See common access-protection profile cap_rights(), 179 cap_rights_limit system call, 178 Capsicum, 30-32, 40, 149, 151, 174-181, 420,810 caught signal, 28, 132, 810 CCB. See common access-method control block CD-ROM, 11, 44, 357-358, 399 CD9660 filesystem, 358, 781

850

cdevsw structure, 368, 371, 373 character device, 370-374, 811 driver. 371 interface, 368, 371, 373-374 ioctl, 374 operations, 373 character-oriented device, 373-374 chdir system call, 46, 813 checksum, 651, 672, 723-725, 741, 746, 811 chflags system call, 439, 481 chgrp system call, 155 child process, 27, 96, 126, 811 chkdq(), 453-454, 506chmod system call, 47, 164, 169, 171, 201, 481 Chorus operating system, 22 chown system call, 47, 155, 164, 481 chroot system call, 33, 46, 180-181, 834 chunk, SCTP, 762-766 CIDR. See classless inter-domain routing CIFS. See common Internet filesystem cipher, AES block, 213, 215 classless inter-domain routing, 652-653, 659,662 CLEAN buffer list, 350, 359 client ID, 575, 811 client process, 50, 811 client server application, 50 interaction, NFS, 562-564 model, 612 programming, 596 clock alternate, 67 interrupt handling, 65-67 interrupt rate, 67 real-time, 58, 795-796 close, device, 391 close-on-exec, 319-321 close system call, 39, 319, 323, 345, 352, 385, 566, 599, 620-621, 700, 741 close vnode operator, 432 closedir(), 445 clustering block, 498, 505-507, 514-517, 811 page, 268, 280, 294, 309, 811 cold start, 811 common access method, 24, 364-366, 393, 399-402, 404, 429 control block, 399-402

layer, 399-402 SCSI I/O request, 400-402 transport, 400 common access-protection profile, 201 common Internet filesystem, 162, 171, 552 communication domain, 50, 594, 606-608, 811 data structures, 608 communication protocol. See protocol compartmentalization, application, 30, 149, 151, 174–175, 808 COMPLETE flag, 466, 470, 472-473 definition, 466 composition, MAC policy, 194-195 Computer Systems Research Group, xxii, xxix, 3, 7-16 condition variables, 112 config, 404, 408, 800, 811 configuration file, 800, 811 kernel, 800-801 network device, 379-380 congestion control network buffering, 643-644 TCP, 752-761 congestion window, 754, 757 connect request, 628, 811 connect system call, 182, 597, 612, 614, 664, 697, 723-725, 736, 767, 811 definition, 597 connection queueing, socket, 610, 613 setup, TCP, 727-728, 736-740 shutdown, TCP, 729, 740-741 states, TCP, 727-730 console, 777, 779, 781, 785, 790, 798-799 serial, 777, 779, 799 contents update, filestore, 497 context switching, 63, 90, 99-114, 812 involuntary, 99, 116 low-level, 100 thread state, 100 voluntary, 99, 101-106 continuation style, 698, 812 control, network device, 379-380 control-output routine, protocol, 630-631 control request, 629, 812 controlling process, 136, 138, 812 controlling terminal, 29, 137-138, 812 revocation of, 345

cooked mode, 383 copy object, 4.4BSD, 258 copy-on-write, 6, 37, 261, 309, 812 core file, 28, 130, 812 coredump(), 136cpu_exit(), 129 cpuid instruction, 422 cpu_mp_announce(), 790 cpu_search(), 123-124 cpuset, 124 cpu_set_fork_handler(), 793 cpu_startup(), 789 cpu switch(), 116 crash dump, 99, 369, 375, 801-803, 812, 828 crash recovery, NFS, 584-586 crash, system, 47-48, 322, 324, 348, 375, 405, 454, 459, 461, 463, 480, 486, 501, 518, 556-557, 561-563, 566-567, 587, 727, 730, 734, 776, 799, 802-803, 812 create vnode operator, 432-433 create_init(), 793 creation and deletion, filestore, 497 credential.95 process, 31, 34-35, 127, 144, 150-157, 179, 181-182, 201-204, 259, 354-355, 556, 564, 793, 800, 803, 830 structure, 152, 160, 179, 181 critical_enter(), 107 critical_exit(), 107 cron, 799 crypto_done(), 208 crypto_freesession(), 207 cryptographic descriptor, 207 framework, 30-31, 35-36, 149, 206-208 verified boot, 777 cryptography asymmetric, 206, 700, 808 session, 206 symmetric, 206, 700, 841 crypto_invoke(), 208 crypto_newsession(), 206–207 crypto_proc(), 208 crypto_register(), 207 csh shell, 139 CSRG. See Computer Systems Research Group ctfconvert. 80

CTSS operating system, 4 cubic congestion-control algorithm, 758, 760 current working directory, 46, 449, 800, 813 *cursig*(), 133, 135 cylinder group, 48, 502

D

D programming language, 78, 80, 188-189 DAC. See discretionary access control dadone(), 402daemon, 813 NFS, 559-562 operation of the, pageout, 292-295 pageout, 58, 92, 233, 236, 238, 240-241, 244, 248, 268, 271-273, 275, 290-297, 307-309, 482, 813, 822, 827-828 process, 324, 813 routing, 684 DARPA. See Defense Advanced Research Projects Agency dastart(), 401 dastrategy(), 401data segment, 36, 69, 71, 263, 813 expansion, 263 data structures autoconfiguration, 407-410 communication domain, 608 interprocess communication, 606-612 socket, 608-611 data transfer, SCTP, 764-766 datagram socket, 595, 813 datalink layer, 622, 813 dead filesystem, 345 deadlock avoidance during fork system call, 260 avoidance when locking resources, 112-114, 335-336, 647 detection, 456, 519 memory, 259, 274-275, 295-296 network, 324-325 prevention, witness, 109, 112-114 snapshot, 485-486 debugger, kernel, 779, 782, 785, 789, 791, 802-803 debugging information in exec header, 71 kgdb, 802 lldb, 142

process, 134, 142-144, 161, 182 system, 802-803 see also ptrace system call decapsulation, 622, 635, 813 decision, local-remote, 678 deduplication, ZFS, 545-546 default pager, 272 Defense Advanced Research Projects Agency, 6, 8, 650, 813 steering committee, 7 definition ATTACHED flag, 466 COMPLETE flag, 466 DEPCOMPLETE flag, 466 defrtrlist_update(), 667 delayed write, 348, 460 delegation, 579, 813 Delta-t, 761 demand paging. See *paging* denial-of-service attack, 739, 813 DEPCOMPLETE flag, 466-467, 470, 472-473.475 definition, 466 dependencies kernel-module, 776 soft updates, 460-464 virtual memory machine, 298-308 descriptor, 39, 813 allocation, 614 duplication, 320-321 management of, 41-42, 316-321 multiplexing, 324-327 table, 40, 316, 814 table, local, 791 use of, 39-41 design 4.2BSD IPC, 8 FreeBSD IPC, 594, 599 I/O system, 39-44 mbuf, 604-605 memory-management, 36-38 NFS, 552-553 /dev, 42, 334, 366-368, 408, 428, 794 filesystem, 44 operation of, 366-367 /dev/console, 798 /dev/cu, 368 /dev/fd, 358 /dev/kmem, 440, 803

/dev/mem, 370, 374, 440 /dev/netmap. 712 /dev/null. 370 /dev/pts, 383 /dev/random, 35 devclass, 408 devd. 413 development model, FreeBSD, 14-17 DEVFS. See device filesystem device, 42, 44, 408 boot, 776-777, 783 character-oriented, 373-374 close, 391 configuration, 402-413 enumeration, 777 identification, 405-407 interrupt handler, 64 module initialization, 794-796 overview, 361-367 pager, 248, 270-271 probing, 369, 405 raw, 372-373 special file, 42, 814 swap, 225, 841 device driver, 30, 42, 368, 777, 782, 787, 793-796,814 attach routine, 405, 407 back ends, 414-428 bottom half of, 369 front ends, 414-428 interrupt handling, 370 maximum transfer size, 371 probe-routine, 405-407 sections of a. 368 support for autoconfiguration, 369, 403-413 support for select system call, 327, 374 top half of, 369 device filesystem, 316, 366–368, 383, 393, 794-795.797 device_attach(), 406-407 device_identify(), 406 device_probe(), 406-407 devices, network, 378-382 device_t, 367 devinfo, 410, 429 dev_t, 367 df, 479 diagnostics, boot-time, 776

diradd structure, 468, 472-476 DIRCHG flag, 476 direct block dependencies, soft updates, 469-470 direct dispatch, 397, 673 direct map, 228 direct memory access, 282, 373, 381, 399, 401, 414, 419, 427-429, 523, 814 direct route, 677 directly-mapped region, 783-784 directory, 45, 443, 814 dependencies, soft updates, 472-476 entry, 45, 434, 814 offset cache, 446-447 operations, 46-47 space allocation, 444-445 structure, 444-447 table, 298, 814 dirrem structure, 464, 475-476, 495-496 DIRTY buffer list, 350, 359 discretionary access control, 32, 149-150, 161-174, 184, 217, 814 disk device, 374-377 interface, 374-375 operations, 374 disk label, 376-377 disk, memory, 780 disk partition, 376, 498, 777-782, 814 disk structure, FFS, 502-504 disk subsystem, 364-366 disk write, ZFS, 536-538 disksort(), 375-376, 401 algorithm for, 376 distributed filesystem, 47 distributed program, 593 DMA. See direct memory access DMU. See zettabyte-filesystem datamanagement unit dnode, 528, 815 structure, 538 ZFS, 528-529 DNS. See domain name system doadump(), 802domain, 815 and type enforcement, 186 name system, 665 zero, 420-423, 426, 428 see also communication domain double indirect block, 435, 815, 820

dpcpu_startup(), 789 dquot structure, 452-454 Dragonfly BSD, xxii, 3 DSL. See zettabyte-filesystem dataset and snapshot layer dsl_dataset structure, 533-534, 538, 542, 544 dsl dir structure, 533 DTE. See domain and type enforcement DTrace, 78-82, 188-189, 790-791 dtrace debug init(), 791 dtrace_pops, 80 dummynet, 702, 704-706 dump, 372, 438, 487, 546 live, 487 dumpsys(), 802 dup system call, 41–42, 48, 319–321, 814, 817 implementation of, 320 dup2 system call, 42, 178, 320, 817 duplication, process virtual memory, 260 - 262dynamic inodes, 441-442 dynamic per-CPU allocator, 789

Е

EACCES system error, 173, 193 EAGAIN system error, 128, 320, 335-336, 391, 614, 617-618, 826 ECAPMODE system error, 179 ECMP. See equal-cost multi-path route ECN. See explicit congestion notification ECONNREFUSED system error, 615 effective GID. See effective group identifier effective group identifier, 155, 815 effective UID. See effective user identifier effective user identifier, 132, 155, 815 EFI. See extended-firmware interface Eighth Edition UNIX, 5 EINTR system error, 62, 98, 128 EINVAL system error, 614, 697 elevated privilege, 158 elevator sorting algorithm, 375, 815 ELF, 245, 779, 781, 785-786, 789 executable format, 70 ELOOP system error, 451 Elz, Robert, 9, 451 embedded systems, 775, 778, 781-784, 796

EMPTY buffer list, 350 EMSGSIZE system error, 616 encapsulating-security payload, 663-664, 689-690, 693, 697 encapsulating security protocol, IPSec, 693 encapsulation, 622, 635, 815 encryption full-disk, 206, 209, 212-217 initialization vector, 209, 213, 215 public-key, 206, 832 entry point, MAC, 34, 188-189, 191-194 entry to kernel, 60-61 enumeration, device, 777 environment, kernel, 779, 781, 787, 794 environment, location of process, 72 EPERM system error, 173, 193 epoch, 73 equal-cost multi-path route, 682 erase character, 383, 815 ERESTART system error, 98 errno, 26, 62, 163, 193-194, 694, 816 ESP. See encapsulating-security payload /etc/defaults/rc.conf, 798 /etc/exports, 547, 560 /etc/fstab. 547 /etc/mac.conf, 200 /etc/master.passwd, 800 /etc/rc, 440, 798-799 /etc/rc.conf, 798 /etc/ttys, 798 Ethernet, 6, 52, 623, 650 event audit. 35, 200-205 channel, 422-424, 426-427 handler, 788-797, 801-802, 816 notification, 329-332 port, 426-427 **EVENTHANDLER REGISTER**, 788 exactly once semantics, 576, 816 exchange_id(), 575 exec header, 69 exec system call, 27, 41, 72, 89, 126, 138, 142, 144, 156–157, 161, 181, 232–233, 258, 261-263, 266, 269, 304, 308-309, 319-320, 330, 832, 835, 837 operation of, 262-263 execve system call, 160, 162, 173, 790, 794, 800 exit(), 128, 136

exit system call, 27, 119, 126, 128, 262, 266–267, 330 operation of, 128–129, 266–267 status, 27, 92, 129 explicit congestion notification, 759 explicit privilege, 158, 181–182 exported filesystem services, 343–344 *extattrctl* system call, 482 extended attributes, 436–438 extended-firmware interface, 377 extension header, 663, 695, 816 extent-based allocation, 516–517 external data representation, 554

F

fast filesystem, 163-164, 166, 168, 170, 174, 209, 353, 431-517, 523-525, 527-529, 531, 535-537, 542, 546, 548, 556, 569, 574,779-780,799 32-bit version, 435, 438, 440-441, 445, 447, 502-504, 516-517 64-bit version, 435-436, 438, 440-441, 445-447, 479, 502-504, 516-517 boot blocks, 503-504 cluster map, 515 cylinder group, 502-503, 813 disk structure, 502-504 ffs_balloc(), 506, 511-512 ffs_read(), 505, 515 ffs_realloccg(), 511–512 ffs write(), 506 file block allocation, 507, 511-513, 809 file block extension, 511 file I/O, 505-507 fragment allocation, 512-514 fragment-descriptor table, 513, 817 fragmentation, 504-507 free-space reserve, 441, 507, 519, 818 implementation of, 502-505, 507-517 layout policies, 508-510 local allocation routines, 510-511 organization, 502-504 overview, 45-48 parameterization, 507 redesign, 501-505 storage optimization, 504-507 superblock, 501 fast retransmission, TCP, 756-757

fault rate, 224, 816 fbtp_patchpoint, 81 fbtp_patchval, 81 fbt_probe_t, 81 fbtp_savedval, 81 fchflags system call, 439, 481 fchmod system call, 47, 164, 177, 481 fchmodat system call, 180 fchown system call, 47, 164, 481 fcntl system call, 8, 178, 319-321, 390, 814 fdesc filesystem, 358 fdisk, 778 FDT. See flattened device trees Federal Information Processing Standard, 8 fetch policy, 223, 816 FFS. See fast filesystem fget(), 179 fhopen system call, 481 FIB. See forwarding information base fifo, 40, 316 file, 39, 443, 816 access validation, 164 append-only, 439 control, filesystem, 432 deactivation, 344 descriptor, 32, 48, 153, 161, 175-180, 183, 192-193, 200 descriptor locking, 322-324 executable, 69 flags, 390, 439-441 handle, NFS, 555, 816 hole in, 47, 819 I/O. FFS, 505-507 I/O, user, 499-501 immutable, 439 interpretation, filesystem, 432 management, filesystem, 432 mapping, 264–265 mode, 162, 164 offset, 41, 318, 816 owner, 31-32, 34, 150-152, 154-155, 158, 161-174, 186 permissions, 30-32, 48, 150, 152, 161-174, 186, 816 reclaim, 344-345 file block allocation, FFS, 507, 511-513, 809 allocation, ZFS, 542-543 locality of reference, 509-510 reading, 505

writing, 506 file entry, 318-319, 816 flag, 319, 321 handling during fork system call, 319 implementation of, 319 object oriented, 318, 321 operations, 318 file locking, 319, 322-324, 454-459 implementation of, 323-324, 456-459 NFS, 553 semantics of, 454-456 filecaps structure, 178 file structure, 316, 609, 817 filedesc structure, 316 filename, 45-46, 816 cache, 346-347, 795 negative caching of, 346 whiteout, 356 filestore abstraction, 498-501 attribute update, 497 contents update, 497 creation and deletion, 497 implementation of, 498-501 operations, 497-498 overview, 48 size update, 498 filesystem, 817 3BSD, 501-502, 504, 508-509 4.2BSD, 502 4.3BSD, 342 4.4BSD, 342, 515 attribute manipulation, 432 buffering, 499-501 CD9660, 358, 781 /dev. 44 distributed, 47 fdesc, 358 file control, 432 file interpretation, 432 file management, 432 independent services, 344-351 initialization, 799 interface, 368 layer, 354-355 links, 449-451 linprocfs, 358 Microsoft NTFS, 47, 171 name creation, 432 name deletion, 432

name lookup, 446-447 name translation, 46, 447-449 naming, 443-451 nullfs, 354-355 operations, 431-433 operator, valloc, 497 operator, vfree, 497 operator, vget, 497 portal, 343, 357-358 /proc, 142-144, 358, 831 procfs, 358 quotas, 8, 451-454, 799 snapshot, 480-487 stackable, 352-358 support for multiple, 43-44 umapfs, 354-355, 564 union, 355-357 see also buffer cache, quotas firewall, 31, 184, 701-707, 785 firmware, 775-778, 782-790 First Edition UNIX, 89 first-level bootstrap, 377 first prison, 181, 790-791, 794 first process, 793 fit, best, 234, 512-513 fit, segregated, 234 flattened device trees, 53, 777, 783-784 fletcher4, 546, 549 floating point in the kernel, use of, 736 floating-point unit, 421 flock system call, 553 flow control, network, 643-644 flow control in TCP, 726 foreground process, 138, 141, 387, 809, 817 fork file, 573, 817 fork system call, 4, 27, 41, 48, 89, 94, 96, 119, 126, 128, 138, 142, 144, 155, 177, 179, 181, 258-261, 292, 304, 306, 308-309, 319-320, 330, 811, 817, 828, 830.832 deadlock avoidance during, 260 file entry handling during, 319 implementation of, 259-260 implementation issues, 261 see also process creation fork1(), 793 Forth interpreter, 780 Fortuna, 35, 212 forward, 650, 655, 662, 664, 670, 673-675, 678, 682-683, 752, 754, 817

forward-mapped page table, 282, 817 forwarding information base, 677 forwarding-mechanism, 677 4.0BSD, 6-9, 501 4.1BSD, 6 4.2BSD, xxii, xxix, 6-8, 36, 40, 42-43, 47, 51-52, 54, 71, 227, 322-323, 326, 371, 385, 403, 593, 653, 674 filesystem, 502 IPC design, 8 scheduler, 791 virtual-memory interface, 7 4.3BSD, xxi, xxii, xxix, 6-8, 37, 71, 291, 309-310, 322, 653, 803 filesystem, 342 Reno release, 6-7 Tahoe release, 6-7, 9 4.4BSD, xxi, xxix, 6-7, 13, 267, 804-805 autoconfiguration, 403 buffer cache, 302 copy object, 258 filesystem, 342, 515 Lite, xxii, 7, 13-14 mbuf design, 604 NFS. 551-552 page replacement, 289, 294 stackable filesystem, 352-353 supported architectures, 7 swap out, 296 swap pager, 273 virtual memory, 37 FPU. See floating-point unit fragmentation, FFS, 504-507 framework, cryptographic, 30-31, 35-36, 149,206-208 framework, MAC, 30, 34, 184-200 free(), 38, 241, 309 free list, 248 free page list, 290 freeblks structure, 464, 468, 476, 495-496 FreeBSD boot partition, 779 development model, 14-17 goals, 17 IPC design, 594, 599 kernel, division of software in, 25 portability of, 23 freefile structure, 468, 476, 495-496 freefrag structure, 468, 470

front ends, device driver, 414–428 fsck, 372, 376, 463, 480, 486–490, 492–494, 501, 504, 509–510, 798–799 background, 486 dependencies, soft updates, 480 *fstat* system call, 47, 164, 629 *fsync* dependencies, soft updates, 477–478 *fsync* system call, 253, 340, 348, 359, 436–437, 461–463, 474, 477–478, 482, 493, 501, 507, 514, 518, 538–539, 566 *fsync* vnode operator, 497–498 *ftruncate* system call, 481 full-disk encryption, 206, 209, 212–217 full virtualization, 414 *futimes* system call, 481

G

gateway, 658, 675, 818 handling, 677-679 g_down, 58, 396-397 GELI, 30, 35, 53, 151, 206, 209, 212-217, 780 block protection, 215-216 flags, 214, 216 I/O model, 216 key management, 213-214 keyfile, 213-214 limitations, 216-217 passphrase, 213-214, 216 startup, 214 threat model, 216-217 g_eli_ctl_resume(), 216 g_eli_start(), 216 g_eli_suspend_one(), 216 g_eli_takefirst(), 216 $g_{eli_{taste}}(), 214, 216$ g_eli_worker(), 216 generation number, 556, 818 generator, random-number, 31, 35, 206, 208-212, 790, 793 generator_gate(), 212 generic security-service application-program interface, 206, 209, 584 GENIE operating system, 4 GEOM. See geometry layer geometry layer, 24, 44, 53, 206, 212, 214, 216, 362, 391-399, 401-402, 407, 429, 524-525, 793-794

flags, 178, 397 operation, 396-397 topology, 392-399 getaddrinfo(), 665 getaddrinfo library call definition, 665 getattr vnode operator, 432 getblk(), 350-351 getdirentries system call, 445 getfsstat system call, 344 gethostbyname(), 665 getlogin system call, 800 getnewbuf(), 351 getnewvnode(), 344-346getpeername system call, 599 getrusage system call, 75 getsockname system call, 599 getsockopt system call, 599, 627, 631 gettimeofday system call, 73-74 getty, 798-800 getuid system call, 179 GID. See group identifier g_init(), 793 $g_{io}_{request(), 216}$ gjournal, 398 global page-replacement algorithm, 289, 818 global vnode table, benefit of, 345 globally-unique identifier partition table, 377, 778-779 goals, FreeBSD, 17 GPT. See globally-unique identifier partition table gptboot, 779, 781-782 gptzfsboot, 779 grant table, 423-427 entry, 424-426 reference, 424-427 Greenwich time. See Universal Coordinated Time group identifier, 31, 151–152, 154–157, 160-166, 168, 170-171, 187, 355, 564, 815, 818, 820, 830, 832, 835, 837 use in file-access validation, 164 gsched, 398 gsignal(), 133 GSSAPI. See generic security-service application-program interface gunzip, 175-176 g_up, 58, 396-397 gvirstor, 398

H

H-TCP. 758 half-open connection, 727 halt, 801-802 hammer_time structure, 784 handle_written_inodeblock(), 469 handling, terminal, 382-391 hard limit, 76, 451, 818 hard link, 449, 818 hardclock(), 66-68, 76hardware performance-monitoring counters, 790-791 processor, 790-791 hardware virtual machine, 421-423, 788 hardware_cache_fetch, 281 Harris, Guy, 9 hash anchor table, 284 hash message-authentication code, 210-211, 216, 403, 546, 549 SHA-256, 213, 215 SHA-512, 213, 215 hash, Modulo-N, 682 HAT. See hash anchor table HBA. See host bus adapter header prediction, TCP, 742, 818 heap, 72, 819 heartbeat requests, 767 heartbeat response, 768 heartbeat, SCTP, 767-768 high watermark on, 819 socket, 610, 616, 643 terminal, 388 history of job control, 7 process management, 89 UNIX. 3-7 HMAC. See hash message-authentication code home directory, 46, 819 hop-by-hop option, 664 hop limit, 662, 819 host bus adapter, 365, 415, 418 host cache metrics, TCP, 737 host unreachable, 657 message, 657, 819 HVM. See hardware virtual machine HWPMC. See hardware performancemonitoring counters

hybrid capability system model, 151 hypercall, 414–428, 788 region, 422 hypervisor, 184, 414–428, 788

I

I/O, 820 asynchronous, 320-322, 326 memory management unit, 420 model, GELI, 216 nonblocking, 320, 325, 614, 617, 619, 826 physical, 372-373 queueing, 369 redirection, 41, 821 signal driven, 320, 325, 838 system design, 39-44 tree, root of, 366, 406, 410 types of kernel, 367-368 I/O buffer, 375 I/O stream, 39, 821 I/O vector. 332-333 i386 boot extender, 779, 781 ICMP. See Internet control message protocol icmp_error(), 658 icmp_input(), 658 ICV. See integrity-check value idempotent, 554, 819 identification, device, 405-407 idle loop, 116, 819 process, 58, 793 queue, 819 swap time, 296 threads, 792-793 IEEE. See Institute of Electrical and Electronic Engineers IETF. See Internet Engineering Task Force ifaddr structure, 635, 639 if_data structure, 636 if_input, 381 ifnet, 378-379, 381-382 structure, 419, 635, 637, 639-640 if_output(), 669-670 IGMP. See Internet group-management protocol ignored signal, 28 IKE. See Internet key exchange imgact, 70

immutable file, 439 implementation of ARP. 655-657 buffer cache, 350-351 dup system call, 320 FFS, 502-505, 507-517 file entry, 319 file locking, 323-324, 456-459 filestore, 498-501 fork system call, 259-260 ioctl system call, 321 kernel malloc, 242-243 kevent system call, 329-332 munmap system call, 264–265 NFS, 558-562 pipe, 40 pmap_enter(), 304-305 pmap_remove(), 305 quotas, 451-454 select system call, 327-329 sleep(), 97-98, 101-104 system call, 62-63 uiomove(), 332-333 wakeup(), 104-106 implicit privilege, 157 implicit send, 765 inactive page list, 290, 307 inactive, reclaim from, 294, 833 inactive vnode operator, 344, 346, 432, 443 INADDR_ANY, 182 inbound, 820 IN_CAPABILITY_MODE(), 180 indirdep structure, 470-471 indirect block dependencies, soft updates, 470-472 indirect route, 677 inetsw, 707 init, 28, 57, 97, 161, 189, 292, 439, 782, 793-794, 798-800, 820, 846 *init_dtrace()*, 790 init hwpmc(), 791 initialization filesystem, 799 kernel, 782-783 user-level system, 798-800 virtual memory, 301-303, 308 see also bootstrapping initiate_write_inodeblock(), 469 inode, 339, 433, 498, 519, 820 allocation, 434

cache, 442-443 contents, 433 definition, 433-442 dependencies, soft updates, 467-469 locality of reference, 508 management, 442-443 number, 356, 437, 441-446, 467, 473, 476, 483, 495, 556, 814 inode wait, 468, 470, 476 inodedep structure, 467, 469-475, 496 inodes, dynamic, 441-442 inpcb structure, 722, 736 in pcballoc(), 723in pcbbind(),723 in_pcbconnect(), 723, 736 in_pcbdetach(),725 in_pcblookup(), operation of, 725 input-output memory-management unit, 420, 427-428 in rtalloc ign(), 672 insecure mode, 440 Institute of Electrical and Electronic Engineers, 8, 136, 364-365, 829 integrity-check value, 691-692, 820 Intel virtualization technology, 421-422 intelligent platform-management interface, 363-364 interactive program, 91, 820 Interdata 8/32, 5 interface addresses, network, 635-636 buffer cache, 348-349 capabilities, 380 capabilities, network, 636-639 character device, 368, 371, 373-374 disk device, 374-375 filesystem, 368 mmap system call, 251-253 mutex. 109-110 network device, 378-379 pager, 267-275 protocol-network, 634-643 protocol-protocol, 631-634 queue, 381 routines, network, 639-641 routing-table, 683-684 socket-to-protocol, 626-631 International Organization for Standardization, 8, 823 model, 622, 650

protocol suite, 649 Internet addresses broadcast, 653-654 multicast, 654-655 packet demultiplexing, 721 structure, 612 Internet association, 721-723 Internet control message protocol, 634, 650-651, 657-659, 666, 669-670, 675, 684, 686, 691, 725, 815, 820 interaction with routing, 658 port unreachable message, 725 Internet domain, 6, 50 Internet Engineering Task Force, 552, 664 Internet group-management protocol, 655 Internet key exchange, 693, 820 Internet ports, 721-723 Internet protocol, xxii, 3, 6, 33, 182, 194, 209, 419, 554, 650-658, 686, 715-716, 723-727, 741, 746, 768-769, 821 firewall, 52, 702-706 fragmentation, 554, 650, 652, 672-673 handling of broadcast message, 672 input processing, 673-675 multicast router, 675 options, 651 output processing, 671-673 packet demultiplexing, 723 packet forwarding, 658, 674-675 protocol header, 652 pseudo-header, 741 responsibilities of, 650 routines. 670-675 version 4 addresses, 652 Internet service providers, 33, 51, 180, 652, 660 interpreter, 70, 820 Forth, 780 interprocess access control, 34, 159-161, 182 interprocess communication, 6, 22, 32, 34, 39-42, 51, 76, 129, 140, 150, 157, 162-164, 180-181, 184, 188, 196, 217, 282, 333-335, 415, 593-646, 648, 790, 820 connection setup, 612-615 data structures, 606-612 data transfer, 615-620 design, 4.2BSD, 8 design, FreeBSD, 594, 599

layers, 599-600 local, 333-338 memory management in, 601-606 message queue, 337-338, 593, 647 model of, 593-599 overview, 50-51 receiving data, 617-620 reliable delivery, 616 semaphores, 335-336 shared memory, 250-258, 338 socket shutdown, 620-621 transmitting data, 616-617 virtualization, 182, 184, 644-646 interprocessor interrupt, 124-125, 421-422, 426, 785, 790, 821 interrupt, 821 asynchronous, 58, 60, 99 handler, device, 64 process, kernel, 57 request, 363 synchronous, 60, 99 interrupt handling, 64-65, 782, 790, 792-793,795 clock, 65-67 device driver, 370 interrupt-vector table, 783 interrupted system call, 62-63 interruptible sleep(), 97, 133 interval time, 74 inverted page table, 284 involuntary context switching, 99, 116 ioctl, character device, 374 ioctl system call, 42, 78, 141, 178, 204, 214, 318, 321, 374, 379-380, 385-387, 620, 629, 635, 638, 700, 712, 714-715, 812 implementation of, 321 ioctl vnode operator, 432 IOMMU. See input-output memorymanagement unit iovec structure, 332-333, 821, 835, 844 IP. See Internet protocol ip6_forward(), 702 *ip6_input()*, 702 ip6_output(), 669, 702, 766 IPC. See interprocess communication ipcs, 334 ip_fastforward(), 702 ip_forward(), 699 IPFW. See Internet-protocol firewall

IPI. See interprocessor interrupt *ip_input()*, 699, 702, 707 ipintr(), operation of, 673-675 IPMI. See intelligent platform-management interface *ip_output(*), 671, 699, 702, 746, 766 operation of, 671-673 IPSec, 30, 35, 52, 148-149, 151, 206, 208-209, 626, 660, 671-673, 675, 688-690, 693, 695, 698-700, 716-717, 820-821, 836, 843-844 authentication header, 691 encapsulating security protocol, 693 implementation, 698-700 overview, 689-690 ipsec4_process_packet(), 699 ipsec_common_input(), 699 IPv4, 182-183, 611, 623, 815 IPv6, 50, 182–183, 611, 649, 659–670, 688, 690-691, 695, 698, 716-717, 815, 820-821,826 addresses, 660-662 autoconfiguration, 666-670 introduction, 659-660 packet formats, 662-664 socket API changes, 664-666 IPX. See Xerox network protocols IRQ. See interrupt request ISA bus, 405-406, 413, 784 ISA. See ISA bus iSCSI, 216, 365-366, 525, 792. See also small-computer system interface ISN. See transmission control protocol initial-sequence number ISO. See International Organization for Standardization ISP. See Internet service providers issignal(), operation of, 135 ITS operating system, 7 IV. See encryption initialization vector

J

jail, 30–34, 149, 151, 158, 180–184, 790, 797, 803, 805 ID, 183–184 *jail* system call, 183 *jail_attach* system call, 183 *jail_get* system call, 184 jail_remove system call, 183 jail_set system call, 184, 645 JID. See jail ID JIT. See just-in-time compilation job, 136, 139 job control, 29, 139-141, 821 history of, 7 signals in FreeBSD, 28 terminal driver support for, 387-388, 391 use of process group, 29 journaled soft updates, 487-496 compatibility, 488 future work, 494-495 introduction, 487-488 journal format, 488-489 performance, 493-494 recovery, 492-493 requirements, 489-492 Joy, William, 6 just-in-time compilation, 701

K

KAME, 659 kdump, 78 keepalive packet, 734, 822 keepalive timer, 734, 822 keg allocator, 238 Kerberos, 800 kernel, 22, 822 address space allocation, 233-244, 787 assembly language in the, 25, 61, 116, 370, 782–784, 786 based virtual machine, 415 bottom half of, 59-60, 809 configuration, 800-801 debugger, 779, 782, 785, 789, 791, 802-803 entry to, 60-61 environment, 779, 781, 787, 794 event polling, 325, 822 facilities, 21-23 I/O, types of, 367-368 initialization, 782-783 interrupt process, 57 linker, 785-787, 789 linker classes, 789 loading of, 301 memory allocation, 38-39

862

memory management, 230-244 memory maps, 231-232 mode, 90, 226, 822 module dependencies, 776 module initialization, 785-786 organization, 23-25, 57-62 partitioning, reason for, 22 preemption, 60 process, 57, 786, 792-794, 822 programming interface, 185, 188-189, 191-192, 194, 639, 785 resource allocation, 259-260 return from, 61-62 security level, 439 state, 90, 822 structure of, 22-23 thread initialization, 792-794 top half of, 59-60, 843 trace-entry structure, 83 trace macros, 83 tracing facility, 77-84 kernel malloc, 241-243, 785, 787-788 implementation of, 242-243 requirements, 241-242 kernel_mount(), 794 kevent system call, 325-326, 329-331, 715 implementation of, 329-332 key, 334, 822 management, 693-698 management, GELI, 213-214 socket, 697 keyfile, GELI, 213-214 kgdb, 802 kick_init(), 794 kill character, 383, 822 kill system call, 132, 182 killpg system call, 140, 831 kinfo_proc structure, 804 kmap_alloc_wait(), 233 kmap_free_wakeup(), 233 kmem_free(), 234 kmem_malloc(), 234 knote structure, 330-332 KPI. See kernel programming interface kproc_start(), 786 kqueue, 40, 317-318, 822 structure, 330-332 system call, 32, 40, 259, 326, 330 KTR. See kernel tracing facility

ktrace, 77–78, 791 ktrace system call, 161, 482 ktrace_init(), 791 ktr_entry structure, 83 KVM. See kernel based virtual machine

L

L2ARC. See zettabyte-filesystem level-2 adaptive-replacement cache label structure, 189, 196-197 la_hold structure, 657 LAN. See local-area network lastlog, 800 layer protocols, network, 51-52 layout, virtual memory, 227-228 lchmod system call, 164, 481 lchown system call, 164, 481 lease, 823 NFS, 559, 580-581 least recently used, 249, 454, 794, 823 LFS. See log-structured filesystem /libexec/ld-elf.so, 245 libficl. 780 libmemstat, 804 libpcap, 700 libprocstat, 804 library, bare-metal system, 780 library, shared, 72 libstand, 780 lightweight process, 146 limitations, GELI, 216-217 limits resource, 26, 75-77 in system, 451 line discipline, 383–385, 823 line mode, 383, 823 link count dependencies, soft updates, 478 - 480link layer, 622, 823 path, 634 *link* system call, 46–47, 481. See also filesystem links *link* vnode operator, 432 link_elf_init(), 789 linker, kernel, 785-787, 789 linker sets, 776, 786 linker_init_kernel_modules(), 789 linker_preload(), 789

linker_stop_class_add(), 789 linprocfs filesystem, 358 Linux operating system, xxi, xxii, xxiii, 7, 11, 17, 71, 95, 358, 815 LISP programming language, 6 listen request, 628, 823 *listen* system call, 597, 612–613, 738, 823 definition, 597 Lite, 4.4BSD, xxii, 7, 13-14 live dump, 487 **lldb**, 80, 142 *llentry* structure, 655–656, 670 lle timer structure, 656 *lltable* structure, 655 In hold, 670 loadable kernel modules, 31, 34, 44, 775-776, 779, 781, 783-786, 794-797, 823 local-area network, 148, 364, 380, 568, 641, 690,747-748,777 local descriptor table, 791 local domain, 50, 432, 823 address structure, 612 local IPC, 333-338 local page-replacement algorithm, 289, 823 local-remote decision, 678 locality of reference, 225, 508-510, 823 lock canrecurse flag, 111 lock synchronization, 110-112 lock vnode operator, 432 LOCKED buffer list, 349 locking advisory, 323, 432, 807 file descriptor, 322-324 mandatory, 323, 824 NFS version 4, 581–583 locking resources on a shared-memory multiprocessor, 106-114, 612 locking resources, deadlock avoidance when, 112-114, 335-336, 647 locking semantics of, file, 454-456 lockstat probe macro, 82 locore.S, 783-784 log-structured filesystem, 537-538, 543 logging, ZFS, 538-540 logical block, 498, 824 login, 155-156, 440, 799-800 login name, 137 login shell, 22

LOMAC. See low watermark mandatory access control long-term scheduling, 117 lookup vnode operator, 342-343, 432 lost+found, 480 low-level context switching, 100 low-level scheduling, 114-117 low pin-count interface, 363-364 low watermark, 824 mandatory access control, 186, 199 socket, 610 terminal, 389 lower half terminal input, 390-391 lower half terminal output, 389 LPC. See low pin-count interface LRO. See transmission control protocol large-receive offload LRU. See least recently used ls, 508 lseek system call, 41, 178, 318, 816 lstat system call, 164, 450 lutimes system call, 481

M

MAC. See mandatory access control mac_error_select(), 194 mac_get_fd system call, 200 *mac_get_file* system call, 200 Mach operating system, 7, 22, 37, 227, 258, 267, 273, 299 mac_init(), 791 mac_init_late(), 791 Macklem, Rick, xxvi, 558-559 mac_label_get(), 197 mac_label_set(), 197 mac_late, 189-190 *mac_policy_conf* structure, 190 *mac_policy_ops* structure, 190 mac_policy_register(), 791 *mac_set_fd* system call, 200 *mac_set_file* system call, 200 mac_t, 200 mac test, 199 mac vnode check write(), 193 $m_{adj()}, 606$ magic number, 69, 377, 824 main memory, 221, 824 malloc(), 38, 72, 198, 227–228, 234, 239, 241, 243, 263, 309, 351, 788, 819
management information base, 191, 805 management mode, system, 777 mandatory access control, 30-32, 34, 149-152, 158, 160-161, 184-200, 217, 437, 630, 655, 791, 793, 797, 800, 804, 824-825,836 entry point, 34, 188-189, 191-194 framework, 30, 34, 184-200 framework startup, 189-190 object association, 198 object destruction, 199 policy composition, 194-195 policy lifecycle, 190 policy registration, 190 security label, 34, 152, 186-189, 195-200,836 mandatory locking, 323, 824 mapped object, 228, 824 mapping, physical to virtual, 302-303, 781 mapping structure, 299, 824 maps, kernel memory, 231-232 maps, virtual memory, 231-232 marshalling, 553 of arguments, 553, 824 masked signal, 132 Massachusetts Institute of Technology, 4, 7 master boot record, 377, 392-393, 395-396, 778-779 maxcontig, 515 maximum segment lifetime, 729-730, 769-770, 825. See also 2MSL timer maximum-segment-size option, TCP, 728, 737 maximum transmission unit, 380, 680, 686, 737-738,770,825 maxusers, 603 $mb_alloc(), 605$ MBR. See master boot record mbuf, 601-605, 795, 825 allocation, 605, 795 cluster, 601-606 data structure description, 601-603 design, 604-605 design, 4.4BSD, 604 storage-management algorithm, 605 structure, 197 utility routines, 606 m_copy(), 746 $m_{copydata()}, 746$

 $m_{copym}(), 606$ memcpy(), 389 memory allocation buffer cache, 351 kernel. 38-39 memory deadlock, 259, 274-275, 295-296 memory disk, 780 memory management, 36-39, 221-308 cache design, 280-282 design, 36-38 goals, 221-226 hardware, VAX, 37 in IPC, 601-606 kernel, 230-244 page-table design, 298-299 portability of, 37 system, 221, 825 memory-management unit, 223, 280, 282-284, 298, 301, 307, 427, 825 design, 282-284, 298-299 I/O, 420 memory overlay, 223 memory, process, 222-223 memstat, 804 menu, boot, 778-781 merged from current, 16 message queue, 51, 825 POSIX, 337-338 System V, 337–338, 593 metadata, 253, 348, 351, 357, 395, 459-463, 478, 484–485, 516–517, 825, 844 metrics, route, 680, 686 MFC. See merged from current $m_{free}(), 605$ *m_get()*, 605 *m_hdr* structure, 601 MIB. See management information base Microsoft NTFS filesystem, 47, 171 MINIX operating system, 7 MIPS architecture, 7, 405, 782-784, 790 mi_startup(), 786 *mi_switch()*, 100, 104, 116–117 mkdir structure, 468, 475 system call, 47, 54, 169, 475, 482 vnode operator, 432 MKDIR_BODY flag, 474-475 MKDIR_PARENT flag, 475 *mkfifo* system call, 481

mknod system call, 481 mknod vnode operator, 432 mlock system call, 253, 290, 307 definition of, 253 MLS. See multilevel security mmap system call, 36-38, 72, 228, 251-252, 259, 264, 266, 270, 304, 334, 347, 548, 824 definition of. 251 interface, 251-253 mmap vnode operator, 432 MMU. See memory-management unit modular congestion control, 758 TCP, 758-761 module_init(), 789 Modulo-N hash, 682 MOS. See zettabyte-filesystem meta-object set motivation for select system call, 324-327 mount, 487, 560-561 mount options, 343 mount system call, 44, 182, 352, 355, 357, 373, 560, 799 mountd, 559-561, 564 mountroot, 794 mprotect system call, 252, 266, 306 definition of, 252 mps_complete_command(), 401 mpssas_action(), 401 $mp_start(), 790$ *m_pullup()*, 606, 724, 741 mq_open system call, 318 mq_receive system call, 337–338 mq_send system call, 337 MS-DOS fat filesystem, 552, 777 MS-DOS operating system, 552, 778 msgrcv system call, 337-338 msgsnd system call, 337 MSL. See maximum segment lifetime msleep(). See sleep() msync system call, 253, 268, 271-272 definition of, 253 mtod(), 606MTU. See maximum transmission unit mtx_destroy(), 109-110 mtx_init(), 109-110 *mtx_lock()*, 109 MTX_OWNED flag, 107 MTX_RECURSE flag, 109

mtx_trylock(), 109-110 mtx_unlock(), 110 MTX_UNOWNED flag, 107 multicast, 636 address, router, 667 Internet addresses, 654-655 message, 725 router, IP, 675 Multics operating system, 4, 7 multihoming, SCTP, 766-767 multilevel feedback queue, 125-126, 825 multilevel security, 34, 186-187, 189-190, 195, 200, 825, 836 multiple-root problem, 569, 825 multiprocessor locking resources on a shared-memory, 106-114,612 scheduling, 122-125 startup, 789 virtual memory for a shared-memory, 37 multiprogramming, 90-91 multiuser mode, 798, 801 munlock system call, 253 definition of, 253 munmap system call, 252, 255, 257, 264, 305 definition of, 252 implementation of, 264-265 mutex, 107-110, 788 interface, 109-110 spin, 107, 839 synchronization, 107-110

N

Nagle, John, 747–748 name creation, filesystem, 432 deletion, filesystem, 432 login, 137 lookup, filesystem, 446–447 translation, filesystem, 46, 447–449 named attributes, 573 named object, 248 *namei*(), 180 naming filesystem, 443–451 shared memory, 252 NAT. See *network address translation*

National Bureau of Standards, 8 nd6 na input(), 670 nd6_output(), 669 nd6_output_lle(), 669-670 nd6 timer(), 670 nd_input(), 669 NEEDRESCHED flag, 116, 134 negative caching of filename, 346 neighbor-discovery, 658, 666-670, 826 Net1 release, 7 Net2 release, 7 NetBSD, xxi, xxii, xxvi, 3, 11, 13-14, 342 netfront, 423 netgraph, 707-711 bridge, 708-711 Ethernet, 708-709 netmap, 712-715 netmask, 652, 826 netstat, 636-637, 646 network address translation, 700 buffering, 643-644 byte order, 651, 826 data flow, 623-624 deadlock. 324-325 device configuration, 379-380 device control, 379-380 device interface, 378-379 device reception, 380-381 device transmission, 381-382 devices. 378-382 flow control, 643-644 interrupt service routine, 642, 793 layer, 621-623, 826 layer protocols, 51-52 mask, 826 protocol capabilities, 626 queue limiting, 643 stack virtualization, 33, 180, 184, 644-646, 683, 717, 776, 786, 789, 791, 805 stack virtualization linker set, 645 time protocol, 580, 796 time synchronization, 74, 796 timer, 67, 625 Network Filesystem, 50, 53, 153, 162-163, 166, 171–174, 206, 209, 217, 340, 345, 348, 354-355, 357, 432-433, 525-526, 547, 551-587, 610, 781, 810-811, 813, 816, 818, 826

4.4BSD, 551-552 access control, 573 asynchronous writing, 566 client-server interaction, 562-564 crash recovery, 584-586 daemons, 559-562 delayed writing, 565 design, 552-553 evolution, 567-586 file handle, 555, 816 file locking, 553 hard mount, 562 implementation of, 558–562 interruptible mount, 563 lease, 559, 580-581 lock reclaim, 586 operation, 553-567 overview, 50, 551 procedures, 569, 572-573, 575-581 security issues, 564 session, 576 soft mount, 563 Network Filesystem version 4, 161-174, 817,837 access. 574 access control functions, 173-174 attributes, 572-573 locking, 581-583 namespace, 572 security, 583-584 versus version 3, 568-571 network interface, 31, 414, 419-420, 423, 428,634-643,795 addresses, 635-636 capabilities, 636-639 cards, 53, 414, 419 layer, 826 routines, 639-641 Xen, 427 newblk structure, 467 newbus, 44, 366, 403, 826 newfs, 441-442, 503-504 NFS. See Network Filesystem nfscbd, 575, 578 nfscb_program(), 579 nfsclclient, 576 nfsclient, 575-576 nfsd, 559-561, 563, 579 nfsiod, 562-563

nfsiod_setup(), 562 nfslock structure, 582-583 nfslockfile structure, 582-583 nfslockhash, 582 nfssvc system call, 560 nfssvc_program(), 561 nfsv4_fillattr(), 573 NIC. See network interface cards nice, 28, 75, 120, 297, 826, 831, 836 Ninth Edition UNIX, 5 N:M threading model, 93 nmount(), 561non-volatile random-access memory, 49, 460.526 nonblocking I/O, 320, 325, 614, 617, 619, 826 nonbypassability, 187, 189 nonuniform memory access, 106, 827 Not-Quite Network Filesystem, 559, 567, 823 Novell, 8 NQNFS. See Not-Quite Network Filesystem NTP. See network time protocol nullfs filesystem, 354-355 NUMA. See nonuniform memory access NVRAM. See non-volatile random-access memory

0

O_ASYNC flag, 320 object association, MAC, 198 destruction, MAC, 199 oriented file entry, 318, 321 object, shadow, 230, 248, 254-258, 837 object, virtual memory, 247-250, 845 objset structure, 529, 531, 533-535, 538-539, 542, 546 obtaining BSD, xxvi Olson, Arthur, 9 one-time password in everything, 800 1:1 threading model, 94 open-file entry, 442 open firmware, 777 open source BSD, 9-14 open system call, 32, 39-41, 48, 162, 168, 177, 179, 183, 198, 203, 269, 318–319, 334, 352, 373, 385, 432–433, 442, 449-450, 481, 599, 700, 712, 813

open vnode operator, 432 openat system call, 32, 179-180 OpenBSD, xxi, xxii, xxvi, 3, 11, 14, 206, 697 OpenBSM, 201 opendir(), 445 operation of /dev, 366-367 operation, NFS, 553-567 operations filestore, 497-498 filesystem, 431-433 terminal, 388-391 OPIE. See one-time password in everything optimal replacement policy, 224, 827 organization, FFS, 502-504 orphaned process group, 141, 827 OS/X operating system, Apple, 3, 436, 445 out-of-band data, 617-619, 630, 827 receipt of, 619 transmission of, 616 overlay, 25, 827 memory, 223 ownership bit, 381

P

packet filter, 52, 637, 701-702, 706-707 forwarding, IP, 658, 674-675 fragmentation, 761 normalization, 707 processing frameworks, 700-715 reception, 642-643 scheduler, 705 transmission, 641 packet demultiplexing Internet addresses, 721 IP, 723 page cache, 368, 372 page clustering, 268, 280, 294, 309, 811 page fault, 223, 817, 827-828, 833 page lists, 290 active, 290 cache, 290 free, 290 inactive, 290, 307 wired, 290 page replacement, 6, 224, 289-295 4.4BSD, 289, 294 criterion for, 289-291

in the VMS operating system, 289 page table, 298, 828 forward-mapped, 282, 817 inverted, 284 pages, 298, 828 reverse-mapped, 284, 834 page-table entry, 282-283, 298, 303, 305-306, 308, 828 page usage, 307-308 page, wired, 271-272, 299-300, 302, 307, 846 pagedep structure, 472-476, 495-496 pagein(), 827operation of, 276-280 pageout daemon, 58, 92, 233, 236, 238, 240-241, 244, 248, 268, 271-273, 275, 290-297, 307-309, 482, 813, 822, 827-828 operation of the, 292-295 pageout in progress, 275 pager, 248-249, 828 definition of, 268-269 device, 248, 270-271 interface, 267-275 physical-memory, 272 swap, 248, 272-275 vnode, 248, 269-270 paging, 6, 36, 71, 223-224, 226, 245-247, 249-250, 276-289, 799, 813, 828 parameters, 291 systems, characteristics of, 223 PAM. See pluggable authentication module panic, 801, 828. See also system crash paravirtualization, 184, 414-428, 788 parent directory, 46 parent process, 27, 96, 126, 828 partial fail, 768 partition. See disk partition passphrase, GELI, 213-214, 216 path MTU discovery, 680, 738, 770, 828 pathname, 46, 828 translation, 342-343 paths through network node, 624 PC-BSD, xxiv PC. See personal computer $p_cansched(), 160$ PCB. See protocol control block PCH. See peripheral controller hub PCI. See peripheral-component interconnect

PDP-11, 5, 62, 89-90 PDP-7, 3, 89 per-CPU allocator, dynamic, 789 perfect forward secrecy, 210 performance. See system performance peripheral-component interconnect, 24-25. 53, 363-366, 405-406, 408-410, 414-416, 420, 423, 428, 777 peripheral controller hub, 362-363 permanent kernel modules, 775, 828 persist timer, 734, 829 personal computer, 7, 60-62, 65, 67, 73, 148, 504, 777, 790 architecture, 362-364 stack growth on, 72 PF. See packet filter PF KEY address extension, 695 association extension, 695 base header, 694 PF_KEY_V2, 693 PF_LOCAL, 630 PFS. See perfect forward secrecy $pgo_alloc(), 268$ $pgo_dealloc(), 268$ pgo_getpage(), 272 pgo_getpages(), 268-272 pgo_haspage(), 268, 272 *pgo_init()*, 268 pgo_putpages(), 268, 270-272, 275 PGP. See pretty-good privacy physical block, 499, 829 physical I/O, 372-373 algorithm for, 372 physical mapping, 299, 829 physical-memory pager, 272 physical to virtual mapping, 302-303, 781 physio(), 372-374PIC. See programmable interrupt controller PID. See process identifier ping, 658, 686 pipe, 40-41, 316, 594, 829 audit, 200-205 implementation of, 40 system call, 40-41, 198, 232, 318, 790, 813 pipeline, 29, 41, 829 assured, 176 placement policy, 223, 829

Plan 9.5 platform_start structure, 784 pluggable authentication module, 800 pmap, 299-300, 303-308 functions, 300-301 initialization, 302, 785 module, 229, 299-301, 303-308. 829 structure, 829 pmap_bootstrap(), 300-301 pmap_bootstrap structure, 785 pmap change wiring(), 301, 307 $pmap_clear_modify(), 301, 307$ *pmap_copy_page()*, 301, 308 pmap enter(), 300, 304–306 implementation of, 304-305 pmap_growkernel(), 300, 302 *pmap_init()*, 300–302 pmap_is_modified(), 301, 308 *pmap_pinit()*, 301, 308 pmap_protect(), 301, 304, 306 pmap_qenter(), 300, 305 *pmap_qremove()*, 300, 305 pmap_release(), 301, 308 pmap_remove(), 300, 305-308 implementation of, 305 pmap_remove_all(), 301, 306 pmap_remove_write(), 301, 306 pmap_ts_referenced(), 301, 307-308 pmap_zero_page(), 301, 308 PMBR. See protective MBR pmc_soft_ev_register(), 791 point-to-point protocol, 707-708 policy composition, MAC, 194-195 policy registration, MAC, 190 poll interface, System V, 326 poll system call, 32, 110, 325-327, 329-330, 374, 385, 630, 715, 829 poll vnode operator, 432 pollfd structure, 326 polling I/O, 325, 829 portability of FreeBSD, 23 memory management, 37 Seventh Edition UNIX, 5 portable operating-system interface, xxii, 8, 39-40, 94, 136, 141, 150, 162-163, 166, 168–171, 173–174, 248, 251, 317, 321-323, 334, 336-338, 385-386, 455, 463, 524-525, 528, 780, 829 message queue, 337-338

real-time, 161-174 shared memory, 338 portal filesystem, 343, 357-358 portmap, 559-560 ports, Internet, 721-723 POSIX. See portable operating-system interface postsig(), 133, 135-136 operation of, 136 PPC architecture, 405 PPP. See point-to-point protocol pr_ctlinput(), 633-634, 658, 725 pr ctloutput(), 631, 725 preadv system call, 419 preemption kernel, 60 thread, 117 prefix option, 668 prepaging, 224, 829 preselection, audit, 201, 204 pretty-good privacy, 30, 209 primary address, 767 pr_input(), 632-633 priority inversion, 103, 830 priority propagation, 103, 830 prison, first, 181, 790-791, 794 prison structure, 181, 646 private mapping, 251, 254-256, 830 private memory, 254-256, 258 priv_check(), 32, 158 priv_check_cred(), 158 privilege, 830 model. 30-34, 149-151, 157-159. 181-182,803-805 separation, 174, 830 PRNG. See pseudo-random number generator probe, 79, 830 probe effect, 79, 830 /proc filesystem, 142-144, 358, 831 procctl system call, 259-260 procedures, NFS, 569, 572-573, 575-581 process, 26, 89, 830 address-space management, 228-230 context, 26, 830 creation, 126-128, 258-262 credential, 31, 34-35, 127, 144, 150-157, 179, 181-182, 201-204, 259, 354-355, 556, 564, 793, 800, 803, 830 debugging, 134, 142-144, 161, 182

first, 793 flags, 142 isolation, 149 kernel, 57, 786, 792-794, 822 kernel interrupt, 57 lightweight, 146 memory, 222-223 model, 149, 831 open-file table, 442, 831 profiling, 63, 74 resource accounting, 31, 67, 129, 790, 800 scheduling, 58, 68, 73, 91-92, 160-161, 782, 792, 794, 796 state, change of, 128, 134, 142 state organization, 92-99 structure, 59-60, 90, 94-98, 100, 804, 831 termination, 128-129, 161, 266-267 virtual address space, 245 virtual memory duplication, 260-262 virtual memory resources, 244-250 virtual time, 74 visibility, 34, 160 process group, 29, 136-139, 830 association with, socket, 140, 608 hierarchy, 96 identifier, 137, 320, 608, 831 job-control use of, 29 leader, 137 orphaned, 141, 827 terminal, 140, 387-388, 390 process identifier, 27, 54, 92, 94, 126-128, 137-138, 145, 209, 259, 317, 387, 685, 831 allocation, 127 process management, 26-29, 69-73, 89-144 history of, 89 process priority, 28, 63, 75, 831 calculation of, 67 processor affinity, 117-118, 121, 240, 831 processor group, 117, 831 processor hardware performance monitoring counters, 790-791 processor rings, 149-150 processor-status longword, 60-62 procfs filesystem, 358 procstat, 144, 804 profclock(), 66, 74 profil system call, 85

profiling process, 63, 74 timer, 66, 74 program relocation, 833 programmable interrupt controller, 427 programming language B,4 C, 3-4, 26, 62 D. 78, 80, 188-189 LISP, 6 protect, 260 protected mode, 779, 781 protection, virtual memory map, 306-307 protective MBR, 779 protocol, 51, 811 buffering policy, 643 capabilities, network, 626 communication, 624-626 control block, 722-723, 766 control-output routine, 630-631 switch, 606, 608 switch structure, 624, 831 protocol family, 608, 805, 831 protocol-network interface, 634-643 protocol-protocol interface, 631-634 protocols, network layer, 51-52 protosw structure, 606, 608 $pr_output(), 632$ pr_usrreqs(), 631 ps, 94, 98, 120 pseudo-header, IP, 741 pseudo-random number generator, 209, 212 pseudo-terminal, 23, 29, 346, 367, 382-384, 388-390, 811-812, 832, 838 psignal(), 133-135 operation of, 133-134 PSL. See processor-status longword ps strings structure, 72 PTE. See *page-table entry* pthread model, 94 pthread_create(), 94 ptrace system call, 78, 106, 142-143, 161, 182 public-key encryption, 206, 832 pure demand-paging, 224, 832 push migration, 832 pv_entry structure, 299, 302-303, 305-310 pwrite system call, 177–178 pwritev system call, 177-178, 419

Q

QFQ. See *quick fair queueing* queue, audit, 204–205 queue limiting, network, 643 quick fair queueing, 705 **quotacheck**, 454 *quotactl* system call, 482 **quota.group**, 452 quotas contribution of, 8 format of record, 452 implementation of, 451–454 limits, 451 **quota.user**, 452

R

racct_init(), 790 race condition, 137-138, 250, 279, 324, 479, 832 radix search trie, 680 RAID. See redundant array of inexpensive disks RAIDZ. See zettabyte-filesystem RAIDZ variant of RAID random-number generator, 31, 35, 206, 208-212, 790, 793 random_harvestq_internal(), 211 range lock, System V, 323 rapid connection reuse, 743, 832 raw device, 372-373 interface, 371, 832 raw mode, 384 raw socket, 42, 651, 658, 686-687, 832 control block, 686 input processing, 687 output processing, 687 rctl_init(), 790 rdrand instruction, 35, 210-211 read system call, 37, 39, 43, 51, 143, 179, 209, 212, 318, 327, 333, 347, 352, 359, 385, 390-391, 599, 615, 700, 816, 826, 831.841 read vnode operator, 497 READ_10, 401 readdir(), 445 readdir vnode operator, 432 *readlink* vnode operator, 432 readv system call, 43, 332, 821

real GID. See real group identifier real group identifier, 155, 832 real mode, 779 real-time clock, 58, 795-796 POSIX, 161-174 scheduling, 28, 75, 91, 117, 252-253 timer. 67. 74 real UID. See real user identifier real user identifier, 155-156, 832 reboot, 801-802 reboot system call, 801-802, 805 receive descriptors, 381 ring, 381 stream, ZFS, 546 window, 732, 833, 838 reception, network device, 380-381 reclaim from inactive, 294, 833 reclaim vnode operator, 345, 432, 443 reclamation dependencies, soft updates, 476 recommended attributes, 573 record, audit, 35, 200-205 recv system call, 43, 630 recvfrom system call, 43, 615, 664, 761, 766 recvit(), 615 recvmsg system call, 43, 598, 615, 620, 630, 764 data structures for, 598 red zone, 38, 241, 833 redundant array of inexpensive disks, 46, 49, 53, 362, 392, 394, 526, 529, 540, 547-548.794 reference monitor, 187, 189 reference string, 224, 833 refinement, capability, 176 region, 245, 833 directly-mapped, 783-784 relative pathname, 46, 828, 833 remote filesystem performance, 565-567 remote procedure call, 553-566, 569, 572-573, 575, 577-579, 581-586, 833 remove vnode operator, 432 rename system call, 47, 482, 557 addition of, 47 rename vnode operator, 432 replacement policy, 223, 833 replay protection, 691-692 request for comments, 568, 573, 584, 659, 661, 688, 739-740

required attributes, 573 resident-set size, 290, 833 resource accounting, process, 31, 67, 129, 790, 800 autoconfiguration, 412-413 limits, 26, 75-77 process virtual memory, 244-250 sharing, 106-114 utilization, 75-76 restore, 438, 546 retransmit timer, 733, 738, 834 return from kernel, 61-62 return from system call, 63 reverse-mapped page table, 284, 834 revocation of controlling terminal, 345 revoke system call, 346, 387, 391, 482 rewinddir(), 445 RFC. See request for comments rfork system call, 40, 94, 126 rip input(), 658Ritchie, Dennis, 3-4, 7 rlimit structure, 95 rm, 479 rmdir system call, 47, 475, 478, 482 rmdir vnode operator, 432 root directory, 45, 834 filesystem, 46, 794, 799, 834 of I/O tree, 366, 406, 410 user, 30-31, 33, 151, 154-155, 157, 174, 181, 793, 834, 841 root_hold_token(), 794 root_mount_rel(), 794 round robin, 115, 834 round-trip time, 565, 735, 769 TCP estimation of, 735–736 route black-hole, 680, 809 metrics, 680, 686 structure, 683 weight, 682 routed, 684 router, 675, 834 advertisement, 666 entry, 667 IP multicast, 675 multicast address, 667 solicitation, 667, 834 routing, 675-686 daemon, 684, 813, 834

information protocol, 684 interaction with ICMP, 658 interface, 685-686 lookup, 680-683 mechanism, 677-684, 834 policy, 684, 835 redirect, 683, 835 redirect message, 683 socket, 685 table interface, 683-684 tables, 677-684 types of, 677 RPC. See remote procedure call rpc.lockd, 557, 559, 561-562 rpc.statd, 559, 561-562 rtalloc(), 682-684 rtentry structure, 671, 678, 683 rtfree(), 683 rtprio system call, 97 rtredirect(), 658, 684 RTT. See round-trip time run queue, 96, 114, 835 management of, 115-117 run-to-completion, 642, 835 $rung_add(), 115$ runq_choose(), 115 operation of, 115 runq_remove(), 115

S

SA. See security association SACK. See transmission control protocol selective acknowledgment Samba, 552 sandbox, 149, 151, 174-181, 835 SAS. See serial-attached SCSI SATA. See serial advanced-technology attachment savecore, 802 saved GID, 157, 835 saved UID, 156-157, 835 sbappendstream(), 744, 746 /sbin/init, 793 *sbrk* system call, 72, 259, 263, 819 SC22 WG15 standard, 8 scatter-gather I/O, 43, 54, 332-333, 416, 419, 615, 835 sched_affinity(), 117

sched clock(), 117 sched_getparam system call, 160 sched_lend_user_prio(), 117 sched_pickcpu(), 123 sched setpreempt(), 117 sched setup(), 791 scheduler(), 296-297 scheduler, packet, 705 scheduling, 90, 414, 836 class, 97, 836 long-term, 117 low-level, 114-117 multiprocessor, 122-125 parameters, 26 priority, 97, 836 process, 58, 68, 73, 91-92, 160-161, 782, 792, 794, 796 real-time, 28, 75, 91, 117, 252-253 short-term algorithm, 126 thread, 106, 114-126 timeshare, 117-126 traditional, 125-126 scripts, user-level startup, 782 SCSI. See small-computer system interface SCTP. See stream control transmission protocol sctp_bindx(), 767 sctp_connectx(), 767 SDT. See statically defined tracepoints secondary storage, 221, 836 secure mode, 440 securelevel, 161 security, 688-700 association, 689, 691, 693-694, 697, 699, 836 association, transport mode, 689 association, tunnel mode, 690 clearance, 186 event auditing, 30-31, 35, 149, 151, 200-205, 792-793, 800, 836 flavor, 584 introduction, 688 issues, NFS, 564 label, MAC, 34, 152, 186-189, 195-200, 836 level, kernel, 439 localization, 30 parameter index, 689-691, 697, 699, 836 protocols, 690-693

protocols implementation, 698-700 triple, 584 seekdir(), 445 see_other_gids, 160 see other uids, 160 segment, 36, 69, 726, 836 bss, 69, 263, 784, 809 data, 36, 69, 71, 263, 813 stack. 36, 69, 263, 839 text, 36, 69, 71, 263, 842 segregated fit, 234 select system call, 32, 325-327, 329-330, 359, 374, 610, 630, 715, 739, 829 device driver support for, 327, 374 implementation of, 327-329 motivation for, 324-327 selfd structure, 327-329 selinfo structure, 327-329 seltd structure, 327-329 seltrue(), 374 selwakeup(), 329, 389 semaphores, 51, 593, 836 System V, 333, 335 virtual memory, 251 semctl system call, 648 semget system call, 336, 648 semop system call, 336, 648 sem_open system call, 317, 336 sem_post system call, 336 sem_wait system call, 336 send stream, ZFS, 546 send system call, 43, 51, 609, 694, 724, 765 send window, 731, 837 sendfile system call, 38, 548 sendit(), 615-616 sendmsg system call, 43, 598, 615, 628, 723 data structures for, 598 sendsig(), 136 sendto system call, 43, 615, 628, 664, 723, 761,764-765 sense request, 629, 837 sequence numbers, TCP, 726 space, 726, 837 variables, TCP, 730-732 sequenced-packet protocol, 761 sequenced-packet socket, 595, 837 serial advanced-technology attachment, 363, 365-366, 399, 402, 405, 407

serial-attached SCSI, 365, 402 serial console, 777, 779, 799 server message block, 525, 552 server process, 50, 837 service location, 559 session, 29, 136-139, 387-388, 837 ID, 575, 837 leader, 138, 837 set-group-identifier program, 155, 837 set-user-identifier program, 155, 533, 837 setattr vnode operator, 432 seteuid system call, 157 setfd structure, 329 setgid binary, 31 setgid system call, 155 setlogin system call, 800 setpgid system call, 137-138 setpriority system call, 831 setrunnable(), 100, 134 setsid system call, 138 setsockopt system call, 599, 621, 627, 631, 654,748,812 settimeofday system call, 74 setuid, 132 binary, 31, 151, 181 system call, 152, 155 Seventh Edition UNIX, 5 portability of, 5 sh shell, 70, 798 SHA. See hash message-authentication code shadow object, 230, 248, 254-258, 837 chain, 255-258 collapse, 257-258 share deny, 581 share reservation, 581 shared library, 72 shared mapping, 36, 251, 253-254, 837 shared memory, 51, 250-258, 593, 838 naming, 252 POSIX, 338 System V, 252, 272, 338 shared text segment, 6 sharing, resource, 106-114 shell, 838 csh, 139 login, 22 sh, 70, 798 shmat system call, 338 shmdt system call, 338, 647

shmem system call, 248, 252, 338 shmget system call, 338 shm_open system call, 179-180, 317, 338 shm_unlink system call, 338 short-term-scheduling algorithm, 126, 838 shutdown, system, 52-54, 801-802 shutdown system call, 599, 619, 740 shutdown final, 802 shutdown_post_sync, 802 shutdown_pre_sync, 802 sigaction system call, 130, 136, 810 SIGALRM, 74 sigaltstack system call, 132 SIGCHLD, 134, 138, 142 sigcode(), 136 SIGCONT, 132, 134, 160, 812 SIGHUP, 141, 387 SIGINT, 136 SIGIO, 320, 390, 608-609, 838 SIGKILL, 28, 132, 134, 160 signal, 28-29, 34, 94-95, 129-141, 160, 838 checking for a pending, 63 comparison with other systems, 129 delivering, 135-136 driven I/O, 320, 325, 838 handler, 28, 130-132, 838 masked, 132 posting, 63-64, 98, 132-134, 142, 838 priority, 28 restrictions on posting, 132 stack, 28, 132 system call, 179 trampoline code, 136, 838 sigprocmask system call, 132, 824 SIGPROF, 74, 85 sigreturn system call, 132, 135–136, 838 SIGSTOP, 28, 132, 144 sigsuspend system call, 101, 132 SIGTHR, 160 sigtramp(), 135 SIGTRAP, 142, 144 SIGTSTP, 145, 391 SIGTTIN, 141, 390 SIGTTOU, 134, 141, 388 SIGURG, 608 SIGVTALRM, 74 silly-window syndrome, 746, 838 TCP handling of, 746-747 single indirect block, 435, 820, 838

single-user mode, 779, 781, 798, 801 Sixth Edition UNIX, 4-5 size update, filestore, 498 slab allocator, 236-237, 787, 791 sleep(), 98, 100-101, 116, 132-133, 292, 369, 615, 838, 843 implementation of, 97-98, 101-104 interruptible, 97, 133 operation of, 104 use of *sleep()*, 97–98, 101 sleep queue, 96, 838 sleepqueue structure, 103-105, 115 sliding-window scheme, 726, 838 slow-start algorithm, TCP, 752-756 small-computer system interface, 363-365, 399-402, 405, 415, 418 bus, 404 I/O request, CAM, 400-402 small-packet avoidance, 747, 839 TCP implementation of, 747–748 SMB. See server message block SMP. See symmetric multiprocessing snapshot, 48, 480, 839 on a large filesystem, 484-486 creating a, 481-483 deadlock, 485-486 maintaining a, 483-484 user visible, 487 ZFS, 541-542 socantrcvmore(), 745 sockaddr structure, 611, 628, 630, 635, 641, 658,677,686,695 sockaddr_dl structure, 635-636 sockaddr_in structure, 182, 664, 724 sockaddr in6 structure, 664 socket, 40, 42-43, 50, 316, 368, 595, 839 address structure, 182, 596, 611, 839 connection queueing, 610, 613 data buffering, 609, 616, 618 data structures, 608-611 error handling, 615 low watermark, 610 options, 626 process group association with, 140, 608 shutdown, 620-621 state transitions during rendezvous, 613 state transitions during shutdown, 621 states, 610 structure, 192

types, 595, 607 using a, 596-599 socket system call, 8, 39-41, 50, 198, 318, 596-597, 606, 612, 628, 631, 713, 761, 813 definition, 596 socket-to-protocol interface, 626-631 socketpair system call, 630, 813 SOCK_SEQPACKET, 761, 765 SOCK STREAM, 596 soconnect(), 614 soft limit, 76, 451, 839 soft link, 449, 839, 841. See also symbolic link soft updates, 48, 459-480, 839 bitmap dependencies, 466-467 dependencies, 460-464 direct block dependencies, 469-470 directory dependencies, 472-476 fsck dependencies, 480 fsync dependencies, 477-478 indirect block dependencies, 470-472 inode dependencies, 467-469 link count dependencies, 478-480 overview, 459-460 reclamation dependencies, 476 structures, 464-466 truncation dependencies, 476 softclock(), 66-69, 74 softdep_disk_io_initiation(), 469 softdep_disk_write_complete(), 469 softdep_update_inodeblock(), 468 software interrupt, 59, 65-66, 210, 839 thread, 65, 208, 793, 795, 839 sohasoutofband(), 744 soisconnected(), 615 soisconnecting(), 736 soisdisconnected(), 745 solid-state disk, 49, 402, 526, 539 solisten(), 613 sonewconn(), 613, 738 soreceive(), 559, 617, 619-620, 647 sorwakeup(), 619 sosend(), 559, 616-617, 647, 746, 768 SPA. See zettabyte filesystem storage-pool allocator SPARC64 architecture, 790 Spec 1170, 8 special file, 42, 316, 839

SPI. See security parameter index spin mutex, 107, 839 split device-driver model, 414-428 SPP. See sequenced-packet protocol SSD. See solid-state disk ssh. 30, 35, 148-149, 209, 383, 798 stack, 839 growth on PC, 72 segment, 36, 69, 263, 839 zero filling of user, 71 stackable filesystem, 352-358 4.4BSD, 352-353 stale data, 565 stale translation, 280-282, 839 standalone device driver, 777, 840 I/O library, 779-780, 840 program, 777, 840 standard error, 41, 840 input, 41, 840 output, 41, 840 start routine, 401, 840 start_init(), 793-794 startup C-language, 784-785 **GELI**, 214 MAC framework, 189-190 multiprocessor, 789 scripts, user-level, 782 system, 52-54, 775-800 witness, 788 stat structure, 318, 439, 629 stat system call, 47, 164, 171, 345, 352, 438, 446,837 statclock(), 66-67, 76 state cookie, 763 stateless protocol, 556, 840 statfs system call, 344 statically defined tracepoints, 80 statistics collection, 66-67, 76 statistics, system, 66-67 sticky bit, 165, 309, 840 stop character, 389 storage-management algorithm, mbuf, 605 strategy(), 351, 401 strategy vnode operator, 469 stream, 762 I/O system, 6 identifier, 764

sequence number, 764 socket, 595, 840 stream control transmission protocol, xxvi, 52, 209, 651, 721, 761-769, 815, 840 association setup, 761-764 association shutdown, 766 chunk, 762-766 data transfer, 764-766 heartbeat, 767-768 multihoming, 766-767 packet header, 762 stream, 764 STREAMS, 326, 632 structures, soft-updates, 464-466 su. 440 Sun Microsystems, 9, 50, 339, 342, 515, 551, 554, 558 superblock, 501, 841 superpages, 284-289, 841 superuser, 46, 154, 158, 322, 803, 805, 841 supplementary group array, 156 suser(), 32 svc_dg_create(), 561 svc_reg(), 561 svc_vc_create(), 561 swap area, 225, 841 device, 225, 841 out, 92, 295-296 out, 4.4BSD, 296 pager, 248, 272-275 pager, 4.4BSD, 273 partitions, 273, 802 space, 225, 272, 841 space management, 273-275 swapin(), 106 operation of, 296-297 swapoff system call, 274 swapper, 296–297, 793, 822, 841 swapping, 36, 73, 225, 295-297, 799, 841 in FreeBSD, reasons for, 295 SWI. See software interrupt swp_pager_async_iodone(), 275 symbolic link, 449-451, 841 symlink system call, 481 symlink vnode operator, 432 symmetric cryptography, 206, 700, 841 symmetric multiprocessing, 106, 841 syn-cache, TCP, 739-740, 762

syn-cookie, TCP, 209, 739-740, 762 sync system call, 341, 482 syncer, 58, 359, 464 synchronization, 81-82, 106-114 lock, 110-112 mutex. 107-110 network time, 74, 796 synchronous interrupt, 60, 99 /sys/kern/sched_4bsd.c, 114 /sys/kern/sched_ule.c, 114 /sys/sys/kernel.h, 786 syscall(), 61 sysctl system call, 84, 122, 160–161, 191, 200, 206, 209, 260, 291, 357, 639, 787, 803-805 SYSINIT, 53, 785-797, 802 syslogd, 799 system activity, 61, 841 system call, 22, 26, 32, 59-60, 150, 152, 200-205, 792, 797-798, 841 handling, 37, 61-63, 100 implementation of, 62-63 result handling, 62-63 return from. 63 system calls accept, 597-598, 611-614, 646, 664, 739 access. 353 adjtime,74 aio_error, 321-322, 330 aio_read, 321, 359 aio_return, 322, 330 aio suspend, 322, 330 aio_waitcomplete, 322 aio_write, 321, 359 audit, 201, 203, 205 bind, 182, 664, 723, 767 cap_enter, 179 cap_getmode, 179 cap_rights_limit, 178 chdir, 46, 813 chflags, 439, 481 chgrp, 155 chmod, 47, 164, 169, 171, 201, 481 chown, 47, 155, 164, 481 chroot, 33, 46, 180-181, 834 close, 39, 319, 323, 345, 352, 385, 566, 599, 620-621, 700, 741 connect, 182, 597, 612, 614, 664, 697, 723-725, 736, 767, 811 dup, 41-42, 48, 319-321, 814, 817

dup2, 42, 178, 320, 817 exec, 41, 72, 89, 126, 138, 144, 156-157, 161, 181, 232-233, 258, 261-263, 266, 269, 304, 308-309, 319-320, 330, 832, 835,837 execve, 160, 162, 173, 790, 794, 800 exit, 27, 119, 126, 128, 262, 266-267, 330 extattrctl, 482 fchflags, 439, 481 fchmod, 47, 164, 177, 481 fchmodat, 180 fchown, 47, 164, 481 fcntl, 8, 178, 319-321, 390, 814 fhopen, 481 flock, 553 fork, 4, 27, 41, 48, 89, 94, 96, 119, 126, 128, 138, 142, 144, 155, 177, 179, 181, 258-261, 292, 304, 306, 308-309, 319-320, 330, 811, 817, 828, 830, 832 fstat, 47, 164, 629 fsync, 253, 340, 348, 359, 436-437, 461-463, 474, 477-478, 482, 493, 501, 507, 514, 518, 538-539, 566 ftruncate, 481 futimes, 481 getdirentries, 445 getfsstat, 344 getlogin, 800 getpeername, 599 getrusage, 75 getsockname, 599 getsockopt, 599, 627, 631 gettimeofday, 73-74 getuid, 179 ioctl, 42, 78, 141, 178, 204, 214, 318, 321, 374, 379–380, 385–387, 620, 629, 635, 638, 700, 712, 714–715, 812 *jail*, 183 jail_attach, 183 *jail_get*, 184 jail_remove, 183 jail_set, 184, 645 kevent, 325-326, 329-331, 715 kill, 132, 182 killpg, 140, 831 kqueue, 32, 40, 259, 326, 330 ktrace, 161, 482 lchmod, 164, 481 lchown, 164, 481 link, 46-47, 481

listen, 597, 612-613, 738, 823 lseek, 41, 178, 318, 816 lstat, 164, 450 lutimes, 481 mac_get_fd, 200 mac_get_file, 200 mac_set_fd, 200 mac set file, 200 mkdir, 47, 54, 169, 475, 482 mkfifo, 481 mknod, 481 mlock, 253, 290, 307 mmap, 36-38, 72, 228, 252, 259, 264, 266, 270, 304, 334, 347, 548, 824 mount, 44, 182, 352, 355, 357, 373, 560, 799 mprotect, 266, 306 $mq_{open}, 318$ mq_receive, 337-338 mg send, 337 msgrcv, 337-338 msgsnd, 337 msync, 253, 268, 271-272 munlock, 253 munmap, 252, 255, 257, 264, 305 nfssvc, 560 open, 32, 39-41, 48, 162, 168, 177, 179, 183, 198, 203, 269, 318-319, 334, 352, 373, 385, 432-433, 442, 450, 481, 599, 700, 712, 813 openat, 32, 179-180 pipe, 40-41, 198, 232, 318, 790, 813 poll, 32, 110, 325–327, 329–330, 374, 385, 630, 715, 829 preadv, 419 procctl, 259-260 profil,85 ptrace, 78, 106, 142-143, 161, 182 pwrite, 177-178 pwritev, 177-178, 419 quotactl, 482 read, 37, 39, 43, 51, 143, 179, 209, 212, 318, 327, 333, 347, 352, 359, 385, 390-391, 599, 615, 700, 816, 826, 831, 841 readv, 43, 332, 821 reboot, 801, 805 recv, 43, 630 recvfrom, 43, 615, 664, 761, 766 recvmsg, 43, 598, 615, 620, 630, 764

rename, 47, 482, 557 revoke, 346, 387, 391, 482 rfork, 40, 94, 126 rmdir, 47, 475, 478, 482 rtprio, 97 sbrk, 72, 259, 263, 819 sched getparam, 160 select, 32, 325-327, 329-330, 359, 374, 610, 630, 715, 739, 829 semctl, 648 semget, 336, 648 semop, 336, 648 sem open, 317, 336 sem post, 336 sem_wait, 336 send, 43, 51, 609, 694, 724, 765 sendfile, 38, 548 sendmsg, 43, 598, 615, 628, 723 sendto, 43, 615, 628, 664, 723, 761, 764-765 seteuid, 157 setgid, 155 setlogin, 800 setpgid, 137-138 setpriority, 831 setsid, 138 setsockopt, 599, 621, 627, 631, 654, 748, 812 settimeofday, 74 setuid, 152, 155 shmat, 338 shmdt, 338, 647 shmem, 248, 252, 338 shmget, 338 shm_open, 179-180, 317, 338 shm_unlink, 338 shutdown, 599, 619, 740 sigaction, 130, 136, 810 sigaltstack, 132 signal, 179 sigprocmask, 132, 824 sigreturn, 132, 135-136, 838 sigsuspend, 101, 132 socket, 8, 39-41, 50, 198, 318, 596-597, 606, 612, 628, 631, 713, 761, 813 socketpair, 630, 813 stat, 164, 171, 345, 352, 438, 446, 837 statfs, 344 swapoff, 274 symlink, 481

system calls (continued) sync, 341, 482 sysctl, 84, 122, 160-161, 191, 200, 206, 209, 260, 291, 357, 639, 787, 804-805 tcsetattr, 815, 822, 846 truncate, 47, 479, 481, 494 umask, 169 undelete, 357, 482 unlink, 46-47, 479, 481, 557 unmount, 352, 482 utimes, 439, 481 vfork, 94, 126, 138, 261-262, 309 wait, 27, 75, 89, 96, 101, 138, 144, 261, 266 - 267wait4, 27, 128-129, 142, 161 write, 26, 37, 39, 42-43, 51, 143, 177, 179, 193, 318, 325, 327, 333, 359, 385, 389, 452, 481, 506, 536, 563, 565-566, 599, 609, 615, 694, 700, 816, 831, 841 writev, 43, 177, 332, 821 system, capability, 174-180, 810 system crash, 47-48, 322, 324, 348, 375, 405, 454, 459, 461, 463, 480, 486, 501, 518, 556-557, 561-563, 566-567, 587, 727, 730, 734, 776, 799, 802–803, 812 system debugging, 802-803 system entry, 58-59 system error EACCES, 173, 193 EAGAIN, 128, 320, 335-336, 391, 614, 617-618,826 ECAPMODE, 179 ECONNREFUSED, 615 EINTR, 62, 98, 128 EINVAL, 614, 697 ELOOP, 451 EMSGSIZE, 616 EPERM, 173, 193 ERESTART, 98 system extended attributes, 168 system library, bare-metal, 780 system management mode, 777 system operation, 800-805 system performance, 62, 65, 69, 71, 73, 91, 116,617 system shutdown, 52-54, 801-802 system startup, 52-54, 775-800 scripts, 798-799 system statistics, 66-67

System V, xxi, 6, 11–12, 95 message queue, 337–338, 593 poll interface, 326 range lock, 323 semaphores, 333, 335 shared memory, 252, 272, 338 terminal driver, 385 SYSUNINIT, 53, 786

Т

tag queueing, 514, 842 tags, 604, 842 tasklist, 464, 470, 476 TCB. See trusted computing base TCP. See transmission control protocol tcp_attach(),736 tcpcb structure, 722, 736 tcp_close(),741 tcp_connect(), 736 tcp_ctloutput(), 748 tcp_delack(),744 tcpdump, 700 *tcp_hc_get()*, 737 tcp_hc_purge(), 738 *tcp_input(*), 732, 742, 745, 757 operation of, 741-745 *tcp_output()*, 732–733, 736, 742, 744–748, 751 operation of, 746 tcp_slowtimo(),733 tcp_usr_send(), 732, 746 tcp_usr_shutdown(), 740 tcsetattr system call, 815, 822, 846 tcsetpgrp(), 141 *tdq_idled()*, 124 TE. See type enforcement telldir(), 445 TENEX operating system, 7 Tenth Edition UNIX, 5 terminal handling, 382-391 low watermark, 389 multiplexer, 368 operations, 388-391 terminal driver, 384-385 bottom half of, 384 close(), 391 data queues, 388-391 functions, 385-387

ioctl(), 385-387 modes, 383-384, 390 open(), 388 special characters, 383 System V, 385 top half of, 384 user interface, 7, 385-387 terminal process group, 140, 387-388, 390 termios structure, 385, 842 text segment, 36, 69, 71, 263, 842. See also shared text segment Thompson, Ken, 3-4, 7, 22 thrashing, 92, 842 thread, 92, 251, 842 preemption, 117 priority, 96-97, 101 priority, calculation of, 105, 117-126 priority, while sleeping, 97 queues, 96 scheduling, 106, 114-126 software interrupt, 65, 208, 793, 795, 839 state, 100 state block, 60, 93, 98, 100, 842 state, change of, 105 structure, 59-60, 90, 98-99, 125, 842 thread exit(), 128 threading model 1:1,94 N:M, 93 threads, idle, 792-793 threat model, GELI, 216-217 three-way handshake, 727-728, 734, 739, 762.842 3BSD.6 filesystem, 501-502, 504, 508-509 tick, 65, 842 time, 65-67 of day, 58, 73 interval, 74 process virtual, 74 quantum, 115, 842 representation, 73 slice, 91, 115, 842-843 stable identifier, 556, 843 synchronization, network, 74, 796 wall clock, 73-74 time zone handling, 9 timeout(), 67-69, 790, 792-793, 795 timer 2MSL, 735, 844

backoff, 734, 843 network, 67, 625 profiling, 66, 74 real-time, 67, 74 resolution of, 73 routines, TCP, 734 virtual-time, 66, 74 watchdog, 67 timeshare scheduling, 117-126 timestamp counter, 422, 790 timestamp option, TCP, 728, 736 timing services, 73-74 TLB. See translation lookaside buffer TLS. See transport-layer security top, 120 top half of device driver, 369 kernel, 59-60, 843 terminal driver, 384 TOPS-20 operating system, 7 trace trap, 142, 843 traced process, 133, 142 tracepoint, 78 track cache, 514-515, 843 tracking file-removal dependencies, 495-496 traditional scheduling, 125-126 trail, audit, 35, 200-205 translation lookaside buffer, 283-288, 304-307, 421, 424, 783, 841, 843 transmission control protocol, xxii, 3, 6, 52, 209, 358, 552, 554, 561, 564, 586, 623, 643-644, 649-651, 664, 671, 689, 691, 693, 698, 704, 707, 721-722, 725-762, 764-765, 767-770, 787, 815, 818-819, 831,842-843 algorithm, 732-741 congestion control, 752-761 connection setup, 727-728, 736-740 connection shutdown, 729, 740-741 connection states, 727-730 data buffering, 754-755 delayed acknowledgments in, 744, 748-749 estimation of round-trip time, 735-736 fast retransmission, 756-757 features of, 725 flow control in, 726 handling of silly-window syndrome, 746-747

transmission control protocol (continued) handling of urgent data, 744 header prediction, 742, 818 host cache, 737 host cache metrics, 737 implementation of small packet avoidance, 747-748 implementation, use of 4BSD, 8 initial-sequence number, 209, 726, 740, 837 input processing, 741-745 large-receive offload, 419 maximum-segment-size option, 728, 737 modular congestion control, 758-761 options, 727 output processing, 745-761 packet header, 727 receive window, 742 retransmission handling, 751–752 segmentation offload, 419, 423, 427 selective acknowledgment, 728, 740, 749-751,764 selective acknowledgment block, 750 send policy, 733, 745-761 sequence numbers, 726 sequence variables, 730-732 slow-start algorithm, 752-756 state diagram, 730 syn-cache, 739-740, 762 syn-cookie, 209, 739-740, 762 timer routines, 734 timers, 733-735 timestamp option, 728, 736 window-scale option, 728 window updates, 748-749 transmission, network device, 381-382 transmission sequence number, 764 transmit descriptor, 382 transmit ring, 382 transport layer, 622, 843 transport-layer security, 30, 149, 209, 688 transport mode, 689, 843 security association, 689 trap(), 61trap handling, 58, 61-62, 64-65, 100 trap type code, 61 TRIM disk advisory, 402 triple indirect block, 435, 820, 843

truncate system call, 47, 479, 481, 494 addition of, 47 truncate vnode operator, 498 truncation dependencies, soft updates, 476 truss. 78 trusted computing base, 30, 147–149, 151, 157-158, 176, 184, 187, 204, 217, 797, 805.844 trusted system, 148 trylock(), 240TSB. See thread state block TSC. See timestamp counter TSN. See transmission sequence number TSO. See transmission control protocol segmentation offload tty driver. See terminal driver ttydevsw, 385, 388 ttydev_write(), 388 ttydisc_close(), 391 ttydisc_getc(), 389, 391 ttydisc_read(), 390-391 ttydisc_reprint(), 389 ttydisc_write(), 389 *ttypoll()*, 385 tunables, 787 tunefs, 487 Tunis operating system, 7, 22 tunnel mode, 623, 632, 690, 844 security association, 690 turnstile, 107, 844 queue, 96, 844 structure, 102-105, 107-108, 112, 115 2MSL timer, 735, 844. See also maximum segment lifetime TXG. See zettabyte filesystem transaction group type-ahead, 383, 844 type enforcement, 34, 186–187

U

U-Boot, 777, 782 uberblock, 523, 527, 532–533, 535, 538–540, 844 **ubldr**, 782 *ucred* structure, 152 UDP. See *user datagram protocol udp_append()*, 725 udp attach(), 723udp_bind(), 723 udp_detach(),725 udp_input(), 724 udp output(),724 $udp_send(), 724$ UEFI. See unified extensible-firmware interface UFS. See fast filesystem UFS1. See fast filesystem, 32-bit version UFS2. See fast filesystem, 64-bit version $ufs_accessx(), 163$ ufs_bmap(), 505, 515 ufs vaccessx(), 168 ugidfw, 187, 200 UID. See user identifier uintptr_t, 197 uio structure, 332-333, 373, 388, 390-391, 497, 765, 844 uiomove(), 333, 373, 389 implementation of, 332-333 ULE scheduling, xxvi, 114, 117-125, 791 umapfs filesystem, 354-355, 564 umask, 164, 170 system call, 169 uma_startup3(),791 uma_timeout(), 791 uma_zalloc(), 244 uma_zcreate(), 244 uma_zfree(), 244 uma_zone_set_max(), 241 undelete system call, 357, 482 unified extensible-firmware interface, 777, 797 union filesystem, 355-357 Universal Coordinated Time, 73-74, 85 universal serial bus, 213, 363-366, 428 universal UID, 208, 423 University of California at Berkeley, 6 UNIX 32V, 5-6 history of, 3-7 Programmer's Manual, 4 Support Group, 5-6 System III, 5-6, 8 System Laboratory, 11-13 System V, 5-6, 8 System V, Release 3, 6 unlink system call, 46-47, 479, 481, 557

unlock vnode operator, 432-433 unmount, 478 unmount system call, 352, 482 unprivileged_proc_debug, 161 update dependency, 461, 844 update vnode operator, 467, 470, 472-473, 476, 497 upper half terminal output, 388-389 urgent data, 320, 726, 733, 746, 844 TCP handling of, 744 transmission, styles of, 617 USB. See universal serial bus use of descriptor, 39-41 USENET, 9, 516 user credential, 152, 844 user datagram protocol, 52, 552-554, 561, 564, 586, 649-651, 689, 691, 693, 698, 707, 721-726, 733, 736, 741, 746, 761, 768-769, 815, 831, 845 control operations, 725 initialization, 723 input, 724-725 output, 724 user file I/O, 499-501 user identifier, 31, 34, 47, 151-152, 154-158, 160-166, 168-173, 187, 204, 354-355, 552, 564, 815, 820, 830, 832, 835, 837, 841, 845 use in file-access validation, 164 user-level startup scripts, 782 user-level system initialization, 798-800 user mode, 90, 226, 845 user-request routine, 622, 625-630, 697, 845 operations, 628-630 USL. See UNIX System Laboratory /usr/bin/login, 799 /usr/local/etc/rc.d, 798 /usr/sbin/config, 429 UTC. See Universal Coordinated Time utimes system call, 439, 481 utmpx, 800 UUID. See universal UID

V

V Kernel operating system, 22 vaccess(), 163–164, 167–168 vaccess_acl_nfs4(), 163, 168, 172–173 vaccess_acl_posixle(), 163, 168

884

valloc filesystem operator, 497 /var/quotas, 452 /var/run/lock. 561 VAX. 5-7 memory management hardware, 37 vegas congestion control algorithm, 759 vfork system call, 94, 126, 138, 261–262, 309 implementation issues, 261 operation of, 262 see also process creation vfree filesystem operator, 497 VFS. See virtual filesystem interface vfs mountroot(), 794 vfs_mountroot_devfs(), 794 vfs_mountroot_shuffle(), 794 vfs_unixify_accmode(), 162 vfs.usermount, 357 vget filesystem operator, 497 vgone(), 345-346 VIMAGE. See network stack virtualization Virtio, Xen, 414-420, 423-424, 427 virtual-address aliasing, 284, 845 virtual address space, 779, 781-784, 787-789, 793, 845 layout of user, 69-73 process, 245 virtual disk, 420, 428 ZFS, 525 virtual filesystem interface, 162-164, 167, 169, 172, 182, 184, 188, 191, 315, 339-344,795 virtual local area network, 180 virtual machine, 32, 130, 545, 700, 845 virtual memory, 6, 30, 149, 414, 419-421, 782, 785, 787-789, 792-794, 796, 804-805,845 4.4BSD, 37 for a shared-memory multiprocessor, 37 advantages of, 225-226 cache alias, 282 cache coherency, 270 change protection, 266 change size, 263-264 data structures, 228-230 duplication, process, 260-262 hardware requirements for, 226 implementation portability, 298-308 initialization, 301-303, 308

interface, 4.2BSD, 7 layout, 227-228 machine dependencies, 298-308 manipulation of, 263-266 map allocation, 304-305 map protection, 306-307 maps, 231-232 object, 247-250, 845 overview, 227-230 resources, process, 244-250 semaphores, 251 usage calculation of, 259-260, 263-264 virtual-network stack allocator, 789 virtual private network, 30, 35, 690, 845 virtual-time timer, 66, 74 virtualization, 32-34, 149-151, 180-184, 414-428,788 IPC, 182, 184, 644-646 **vmcall** instruction, 422 *vm* daemon(), 58, 296 vmem resource allocator, 234-243 vm_fault(), 76, 249, 276-278, 299, 307 vm_forkproc(), 127 vm_ksubmit_init(), 789 *vm_map* structure, 230–232, 254, 280, 300 *vm_map_entry* structure, 230–232, 245–248, 253, 255, 259, 261-262, 264-267, 276, 286, 303, 306, 309 *vm_object* structure, 229–230, 245, 247-250, 253-258, 264-276, 278-280, 287, 294–295, 302–305, 309, 334, 339 vm.overcommit, 260 *vm_page* structure, 230, 237, 247, 249–250, 267-268, 270-271, 285, 301-302, 305, 307-308 vm_page_alloc(), 291 vm_page_io_finish(), 275 vm_pageout(), 292 vm_pageout_scan(), 292–294 vm_pageout_update_period, 295 vm_pager_bufferinit(), 790 vm_page_startup(), 301 vm_page_test_dirty(), 307–308 *vm_pmap* structure, 229–230, 299–302, 306-308 VMS operating system, 7-8, 289 page replacement in the, 289 vmspace structure, 229-230, 245-246, 259, 262

vmspace_exec(), 308 vmspace_fork(), 308 vmspace_free(), 308 vnet structure, 644-646 vnet data startup(), 789 vnlru vnode recvcling daemon, 58 vnode, 43, 316, 339, 609, 845 cache. 249 description of, 339-342 operations, 342 vnode operator access, 432 advlock, 432 blkatoff, 497-498 close, 432 create, 432-433 fsvnc, 497-498 getattr, 432 inactive, 344, 346, 432, 443 ioctl, 432 link, 432 lock, 432 lookup, 342-343, 432 mkdir, 432 mknod, 432 mmap, 432open, 432 poll, 432 read, 497 readdir.432 readlink, 432 reclaim, 345, 432, 443 remove, 432 rename, 432 rmdir, 432 setattr, 432 strategy, 469 symlink, 432 truncate, 498 unlock, 432-433 update, 467, 470, 472-473, 476, 497 write, 497 vnode pager, 248, 269-270 vnode structure, 192-194, 196, 198 vnode_pager_setsize(), 269 vn_write(), 193-194 voluntary context switching, 99, 101-106 VOP_ACCESS(), 162 vop_access_args structure, 353

VOP_ACCESSX(), 162, 168 VOP_ACLCHECK(), 167 VOP_GETACL(), 167 VOP_SETACL(), 167 vop_stdaccessx(), 162 VPN. See virtual private network vring structure, 416 vring_avail structure, 418 vring_desc structure, 418 vring_used structure, 418 vring_used_elem structure, 418 VT. See Intel virtualization technology

W

WAFL. See write-anywhere file-layout filesystem wait channel, 98, 101, 104-105, 111, 846 wait system call, 27, 75, 89, 96, 101, 138, 144, 261, 266-267, 846 wait4 system call, 27, 128-129, 142, 161 operation of, 129 wakeup(), 104-105, 120, 275 implementation of, 104-106 operation of, 105 wakeup one(), 105 wall clock time, 73-74 WAN. See wide-area network watchdog timer, 67 whiteout, filename, 356 wide-area network, 568, 759 wildcard route, 677, 725, 846 window probe, 734, 846 window-scale option, TCP, 728 window system, 140. See also X Window System Windows operating system, xxii wine, xxix wired page, 271-272, 299-300, 302, 307, 846 definition of, 233 list, 290 witness deadlock prevention, 109, 112-114 startup, 788 word-erase character, 383, 846 working set, 225, 846 worklist structure, 464, 467 workstation, 221

write-anywhere file-layout filesystem, 543 write system call, 26, 37, 39, 42–43, 51, 143, 177, 179, 193, 318, 325, 327, 333, 359, 385, 389, 452, 481, 506, 536, 563, 565–566, 599, 609, 615, 694, 700, 816, 831, 841 write vnode operator, 497 writev system call, 43, 177, 332, 821 wrmsr instruction, 422

X

X Display Manager, 799 X/OPEN, 8 X Window System, 748 X.25, 641 XDR. See external data representation Xen, 184, 414-415, 419-427, 788, 790 block interface, 427 network interface, 427 Virtio, 414-420, 423-424, 427 XenBus, 423 xen_hvm_init(), 422 XenStore, 420, 423 Xerox network protocols, 631, 815 xform-switch structure, 699 XINU operating system, 7 XPT. See common access-method transport xpt_action(), 401 $xpt_done(), 402$ xpt_schedule(), 401 xterm, 383

Y

Yarrow, 35, 210-212, 793

Z

zalloc(), 39, 243–244 ZAP. See zettabyte-filesystem attribute processor zero filling of user stack, 71 zettabyte-filesystem, xxvi, 25, 33, 166, 171, 180, 182, 184, 270, 420, 427, 496–497, 523–549, 556, 574, 779–780, 794, 799 attribute processor, 524, 528–529, 533–535, 545–546 block free, 543–545 block pointer, 529–531

data-management unit, 524, 528, 530, 546 dataset and snapshot layer, 524, 527, 529, 533 deduplication, 545-546 design tradeoffs, 546-549 disk write, 536-538 dnode, 528-529 features, 523-527 file block allocation, 542-543 input-output module, 525 intent log, 524, 528, 535, 538 level-2 adaptive-replacement cache, 525, 539 logging, 538-540 meta-object set, 525, 527-528, 531-534, 538-539, 542, 546 objset layer, 534-535 objset structure, 531-532 operation, 535-546 organization, 527-532 overview, 49 POSIX layer, 524, 528, 538 RAIDZ variant of RAID, 524-525, 530-531, 540-541, 543, 547-549, 779 receive stream, 546 remote replication, 546 send stream, 546 snapshot, 541-542 storage-pool allocator, 525, 527-529, 531, 533.542-545 structure, 532-535 transaction group, 530-532, 535-536, 538-539, 543 virtual disk, 525 ZVOL volume, 525, 527-528, 531-536, 539, 542-548, 815 *zfree()*, 39 ZFS. See zettabyte-filesystem ZIL. See *zettabyte-filesystem intent log* zil_header structure, 535 ZIO. See zettabyte-filesystem input-output module znode, 528, 815 zombie process, 96, 128-129, 846 zone allocator, 239-241, 791, 793 zone, red, 38, 241, 833 zones, 239-241, 243-244 ZPL. See zettabyte-filesystem POSIX layer ZVOL. See zettabyte-filesystem ZVOL volume