

The Design and Implementation of the
4.3BSD UNIX[®]
Operating System

Samuel J.
Leffler

Marshall Kirk
McKusick

Michael J.
Karels

John S.
Quarterman



The Design and Implementation of the

4.3BSD UNIX Operating System

The Design and Implementation of the

4.3BSD UNIX Operating System

Samuel J. Leffler
Silicon Graphics, Inc.

Marshall Kirk McKusick
University of California, Berkeley

Michael J. Karels
University of California, Berkeley

John S. Quarterman
Texas Internet Consulting



Addison-Wesley Publishing Company

Reading, Massachusetts • Menlo Park, California • New York
Don Mill, Ontario • Wokingham, England • Amsterdam • Bonn
Sydney • Singapore • Tokyo • Madrid • San Juan

This book is in the Addison-Wesley Series in Computer Science

Michael A. Harrison: Consulting Editor

Keith Wollman: Sponsoring Editor
Carolyn Berry: Marketing Manager
Karen Myer: Production Supervisor
Hugh Crawford: Manufacturing Supervisor
Patsy DuMoulin: Production Coordinator
Jean Seal: Cover Designer
Jean Hammond: Book Designer
Jaap Akkerhuis: Troff Macro Designer
Lyn Dupré: Copy Editor

John Lasseter created the cover art.

UNIX is a registered trademark of AT&T Bell Laboratories in the United States and other countries. 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 Addison-Wesley was aware of a trademark claim, the designations have been printed in initial caps or all caps.

The programs and applications presented in this book have been included for their instructional value. They have been tested with care, but are not guaranteed for any particular purpose. The publisher does not offer any warranties or representations, nor does it accept any liabilities with respect to the programs or applications.

Library of Congress Cataloging-in-Publication Data

The Design and implementation of the 4.3 BSD UNIX operating /
system by Samuel J. Leffler ... [et al.].
p. cm
Includes bibliographies and index.
ISBN 0-201-06196-1
1. UNIX (Computer operating system) I. Leffler, Samuel J.

QA76.76.063D474 1988
005.4'3--dc19

88-22809
CIP

Reprinted with corrections October, 1990

Copyright © 1989 by Addison-Wesley Publishing Company, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America. Published simultaneously in Canada.

8 9 10 HA 95949392

Dedication

This book was inspired by Elliot Organick's book, *The Multics System*.

This book is dedicated to the UNIX community. Without the contributions of its members, there would be nothing about which to write.

Preface

This book is the first authoritative and full-length description of the design and implementation of the research versions of the UNIX system developed at the University of California at Berkeley. Most detail is given about 4.3BSD, which incorporates the improvements of the previous Berkeley versions.

The UNIX System

The UNIX system runs on computers ranging from personal home systems to the largest supercomputers. It is the operating system of choice for most multiprocessor, graphics, and vector-processing systems, and is widely used for its original purpose of timesharing. It is the most portable operating system ever developed. This portability is due partly to its implementation language, C [Kernighan & Ritchie, 1978] (which is itself one of the most widely ported languages), and partly to the elegant design of the system. Many of the system's features are imitated in other systems [O'Dell, 1987].

Since its inception in 1969 [Ritchie & Thompson, 1978], the UNIX system has developed in a number of divergent and rejoining streams. The original developers continue to advance the state of the art with their Ninth Edition UNIX inside AT&T Bell Laboratories. Meanwhile, AT&T licenses UNIX System V as a product. Both of those systems have been strongly influenced by the Berkeley Software Distributions produced by the Computer Systems Research Group (CSRG) of the University of California at Berkeley.

Berkeley Software Distributions

These Berkeley systems have introduced a number of useful programs and facilities to the UNIX community, such as the text editor `vi` in 2BSD (the Berkeley PDP-11 system), demand-paged virtual-memory support in 3BSD (the first Berkeley VAX system), performance improvements in 4.0BSD, job control,

autoconfiguration, and long C identifiers in 4.1BSD, as well as reliable signals, improved networking, sophisticated interprocess-communication (IPC) primitives, a fast filesystem, and more performance improvements in 4.2BSD and 4.3BSD. 4.2BSD and 4.3BSD are the bases for the UNIX systems of many vendors, and are used internally by the development groups of many other vendors. Many of these developments have also been incorporated by System V or added by vendors whose products are otherwise based on System V.

The implementation of the TCP/IP networking protocol suite in 4.2BSD and 4.3BSD and the availability of those systems explain why the TCP/IP networking protocol suite is so widely implemented throughout the world. Numerous vendors have adapted the Berkeley networking implementations, whether their base system is 4.2BSD, 4.3BSD, System V, or even VMS.

4BSD has 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 4.3BSD for POSIX.

Material Covered in this Book

Except in the introductory historical material, the word “UNIX” is used in this book to refer to 4.3BSD. Where particular hardware is relevant, the book refers to the Digital Equipment Corporation (DEC) VAX. Although many vendors sell systems derived from 4.2BSD or 4.3BSD that run on numerous processors and computer systems (such as the Motorola 68020, the National Semiconductor 32032, or the Intel 80386), 4BSD was originally developed on the VAX. Because that machine is one of those on which the current Berkeley software distribution runs, it still provides a convenient point of reference.

This book is about the *internal* structure of 4.3BSD [Quarterman *et al.*, 1985], and about the concepts, data structures, and algorithms used in implementing 4.3BSD's system facilities. Its level of detail is similar to that of Bach's book about System V UNIX [Bach, 1986]; however, this text focuses on the facilities, data structures, and algorithms used in the Berkeley variant of the UNIX operating system. The book covers 4.3BSD 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, memory management, the I/O system, the filesystem, 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. Like Organick's book about Multics [Organick, 1975], it is an in-depth study of a contemporary operating system.

Readers who will benefit from this book include operating-system implementors, system programmers, UNIX application developers, administrators, and curious users. The book may be read as a companion to the source code of the system, falling as it does between the manual [CSRG, 1986] and the code in detail of treatment. But the book is specifically neither a UNIX programming

manual nor a user tutorial (for a tutorial see [Libes & Ressler, 1988]). Familiarity with the use of some version of the UNIX system (see, for example [Kernighan & Pike, 1984]), and with the C programming language (see, for example [Kernighan & Ritchie, 1988]), would be extremely useful.

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 **second-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* [Peterson & Silberschatz, 1985]. Familiarity with the concepts of network protocols will be useful in some of the later chapters [Tanenbaum, 1988; Stallings, 1985; Schwartz, 1987].

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 without any asterisks can be found in the text. Exercises with a single asterisk require a step of reasoning 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 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 is used to display algorithms. Boldface font is used to identify program names and filesystem pathnames. Italics font is used to introduce terms that appear in the glossary, and to identify the **names of system calls**, variables, routines, and structure names. Routine names (other than system calls) are further identified by following the name with a pair of parenthesis (e.g., *malloc()* is the name of a routine while *argv* is the name of a variable).

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

- **Part 1, 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 its research orientation. Chapter 2, *Design Overview of 4.3BSD*, describes the nature of 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. Chapter 3, *Kernel Services*, explains how system calls are performed, and describes in detail some of the basic services of the kernel.
- **Part 2, Processes** The first chapter in this part—Chapter 4, *Process Management*—lays the foundation for subsequent chapters by describing the

structure of a process, the algorithms used for scheduling the execution of processes, and the synchronization mechanisms used by the system to ensure consistent access to kernel-resident data structures. In Chapter 5, *Memory Management*, the virtual-memory-management system is discussed in detail.

- **Part 3, I/O System** First, Chapter 6, *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 three chapters that describe the details of the three main parts of the I/O system. Chapter 7, *The Filesystem*, describes the data structures and algorithms that implement the filesystem. Chapter 8, *Device Drivers*, explains block and character device drivers, including the method of autoconfiguration used by the system to determine what physical devices are attached. A detailed example of a disk device driver is given. Chapter 9, *Terminal Handling*, describes support for character terminals and provides a description of a character-oriented device driver.
- **Part 4, Interprocess Communication** Chapter 10, *Interprocess Communication*, describes the mechanism for providing communication between related or unrelated processes. Chapters 11 and 12, *Network Communication* and *Network Protocols*, are closely related, as the facilities described in the former are largely implemented in terms of specific protocols, such as the TCP/IP protocol suite, described in the latter.
- **Part 5, System Operation** Chapter 13, *System Startup*, discusses system startup, shutdown, and configuration, and 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 1 are relatively independent of one another and may be read separately. Chapter 13 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.

Acknowledgments

We thank the following people, all of whom read and commented on early drafts of the book: Wayne Hathaway (Ultra Network Technologies), Maurice Herlihy (Carnegie-Mellon University), Thomas LeBlanc (University of Rochester), Larry Peterson (University of Arizona), Robert Elz (University of Melbourne), Peter Collinson (University of Kent), Dennis Ritchie (AT&T Bell Laboratories), Matt Koehler (Sun Microsystems), Eric Allman (International Computer Science Institute), Mike Hibler (University of Utah), Robert Gingell (Sun Microsystems), Joseph Moran (Legato Systems), Christopher Torek (University of Maryland), Kevin Fall (University of California, Berkeley), Peter Salus (Usenix Association), Evi Nemeth (University of Colorado), Jim Lawson (Pixar), Mike Paquette (Pixar),

Mike Russell (Pixar), Donn Seeley (University of Utah), Jay Lepreau (University of Utah), Bill Shannon (Sun Microsystems), Donald Coleman (Unisoft Corporation), Michael O'Dell (Prisma, Incorporated), Michel Gien and associates (Chorus Systèmes).

This book was produced using **pic**, **tbl**, **eqn**, and **ditroff**. 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 **cip** to produce **pic** input that had to be hand-tuned to clean up ragged lines and improperly centered text. Figure placement and widow elimination were handled by the **ditroff** macros, but orphan elimination and production of even page bottoms had to be done by hand. In the end, we concluded that there must be a better way!

The authors encourage readers to send suggested improvements or comments about typographical or other errors found in the book by electronic mail to bsdbook-bugs@berkeley.edu.

References

- Bach, 1986.
M. J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, Englewood Cliffs, NJ (1986).
- Bentley & Kernighan, 1986.
J. Bentley & B. Kernighan, "Tools for Printing Indexes," Computing Science Technical Report 128, AT&T Bell Laboratories, Murray Hill, NJ (1986).
- CSRG, 1986.
CSRG, "UNIX Programmer's Manual, 4.3 Berkeley Software Distribution, Virtual VAX-11 Version," Six Volumes and an Index Volume, University of California Computer Systems Research Group, Berkeley, CA (April 1986).
- Kernighan & Ritchie, 1978.
B. W. Kernighan & D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ (1978).
- Kernighan & Pike, 1984.
B. W. Kernighan & R. Pike, *The UNIX Programming Environment*, Prentice-Hall, Englewood Cliffs, NJ (1984).
- Kernighan & Ritchie, 1988.
B. W. Kernighan & D. M. Ritchie, *The C Programming Language*, 2nd ed, Prentice-Hall, Englewood Cliffs, NJ (1988).
- Libes & Ressler, 1988.
D. Libes & S. Ressler, *Life with UNIX*, Prentice-Hall, Englewood Cliffs, NJ (1988).
- O'Dell, 1987.
M. O'Dell, "UNIX: The World View," *Proceedings of the 1987 Winter USENIX Conference*, pp. 35-45 (January 1987).

Organick, 1975.

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

Peterson & Silberschatz, 1985.

J. Peterson & A. Silberschatz, *Operating System Concepts*, Addison-Wesley, Reading, MA (1985).

Quarterman *et al.*, 1985.

J. S. Quarterman, A. Silberschatz, & J. L. Peterson, "4.2BSD and 4.3BSD as Examples of the UNIX System," *ACM Computing Surveys* **17**(4), pp. 379–418 (December 1985).

Ritchie & Thompson, 1978.

D. M. Ritchie & K. Thompson, "The UNIX Time-Sharing System," *Bell System Technical Journal* **57**(6 Part 2), pp. 1905–1929 (July-August 1978). The original version [*Comm. ACM* **7**(7), pp. 365–375 (July 1974)] described the 6th edition; this citation describes the 7th edition.

Schwartz, 1987.

M. Schwartz, *Telecommunication Networks*, Addison-Wesley, Reading, MA (1987).

Stallings, 1985.

R. Stallings, *Data and Computer Communications*, Macmillan, New York, NY (1985).

Tanenbaum, 1988.

A. S. Tanenbaum, *Computer Networks*, 2nd ed, Prentice-Hall, Englewood Cliffs, NJ (1988).

Contents

Part 1	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	6
	Other Organizations	7
	Berkeley Software Distributions	7
	UNIX in the World	8
1.2	BSD and Other Systems	8
	The Influence of the User Community	9
1.3	Design Goals of 4BSD	10
	4.2BSD Design Goals	11
	4.3BSD Design Goals	12
	Future Berkeley Releases	12
1.4	Release Engineering	14
	References	15
Chapter 2	Design Overview of 4.3BSD	19
2.1	UNIX Facilities and the Kernel	19
	The Kernel	20
2.2	Kernel Organization	20
2.3	Kernel Services	23
2.4	Process Management	23
	Signals	25
	Process Groups	26

2.5	Memory Management	26
	BSD Memory-Management Design Decisions	27
	Memory Management Inside the Kernel	28
2.6	I/O System	29
	Descriptors and I/O	30
	Descriptor Management	31
	Files	32
	Devices	33
	Socket IPC	33
	Scatter/Gather I/O	34
2.7	Filesystem	34
2.8	Devices	37
2.9	Terminals	38
2.10	Interprocess Communication	38
2.11	Network Communication	39
2.12	Network Implementation	40
2.13	System Operation	40
	Exercises	41
	References	41

Chapter 3 Kernel Services

43

3.1	Kernel Organization	43
	System Activities	43
	Run-Time Organization	44
	System Processes	45
	Entry to the Kernel	46
	Return from the Kernel	47
3.2	System Calls	47
	Result Handling	47
	Returning from a System Call	48
3.3	Traps and Interrupts	49
	I/O Device Interrupts	49
	Software Interrupts	50
3.4	Clock Interrupts	50
	Timeouts	51
	Process Scheduling	53
3.5	Timing	53
	Real Time	53
	Adjusting the Time	53
	External Representation	54
	Interval Time	55
	Profiling	55
3.6	Process Management	55
3.7	User and Group Identifiers	58
	Host Identifier	60
3.8	Resource Controls	60
	Process Priorities	60

Resource Utilization	61
Resource Limits	61
Filesystem Quotas	62
3.9 System Operation	62
Accounting	62
Exercises	63
References	64

Part 2 Processes

67

Chapter 4 Process Management

69

4.1 Introduction	69
Multiprogramming	70
Scheduling	71
4.2 Process State	72
The Proc Structure	72
The User Structure	77
Memory	79
The Text Structure	79
4.3 Context Switching	79
Process State	80
Low-Level Context Switching	80
Voluntary Context Switching	81
Intraprocess Context Switching	83
Synchronization	84
4.4 Process Scheduling	86
Calculations of Process Priority	87
Process-Priority Routines	88
Process Run Queues and Context Switching	89
4.5 Process Creation	91
4.6 Process Termination	93
4.7 Signals	94
Process Groups	96
Comparison with Other Systems	97
Changes to 4.3BSD Signals in POSIX	99
Posting a Signal	99
Delivering a Signal	101
Job Control	102
4.8 Process Debugging	103
Exercises	105
References	107

Chapter 5 Memory Management

109

5.1 Terminology	109
Processes and Memory	110

	Paging	111
	Replacement Algorithms	112
	Working-Set Model	113
	Swapping	113
	Secondary Storage	114
	Advantages of Virtual Memory	114
	Hardware Requirements for Virtual Memory	114
5.2	Evolution of 4.3BSD Memory Management	115
	Version 7 UNIX	115
	UNIX 32V	116
	3BSD	116
	4.1BSD	117
	4.3BSD	118
5.3	VAX Memory-Management Hardware	118
	VAX Virtual Address Space	118
	VAX Page Tables	119
	System-Address Translation	120
	User-Address Translation	121
	Page Faults	122
	Translation Buffers	124
5.4	Management of Main Memory: The Core Map	124
	Physical-to-Virtual Translation	126
	Memory Free List	126
	Synchronization	126
	Text-Page Cache	127
	Core-Map Limits	127
	Memory-Allocation Routines	127
5.5	Management of Swap Space	128
5.6	Per-Process Resources	129
	4.3BSD Process Virtual Address Space	129
	Page Tables	129
	Types of Page-Table Entries	131
	Modified Pages	134
	Text Page Tables	134
	Swap Space	134
5.7	Creation of a New Process	136
	Duplicating Kernel Resources	136
	Duplicating the User Address Space	137
	Implementation Issues	138
	Creating a New Process Without Copying	138
5.8	Execution of a File	140
5.9	Change Process Size	142
5.10	Termination of a Process	144
5.11	Demand Paging	145
	Fill-on-Demand Pages	146
	Fill-on-Demand Klustering	146
	Interaction with the Filesystem Cache	147
	Pagein of Swapped Pages	147

5.12	Page Replacement	149
	Global CLOCK Algorithm	150
	The Paging Daemon	151
	Paging Parameters	151
	Two-Handed Clock	153
	Operation of Pageout	153
5.13	Swapping	155
	The Swapping Process	156
	Choosing a Process to Swap In	156
	Involuntary Swapping	157
	Choosing a Process to Swap Out	158
	Swapout	158
	Swapin	159
	Swapping of Text Images	160
	Exercises	161
	References	163

Part 3 I/O System

167

Chapter 6 I/O System Overview

169

6.1	I/O Mapping from User to Device	169
	Character Devices	170
	Block Devices	171
	Socket-Interface Buffering	172
6.2	Descriptor Management and Services	172
	Open File Table	173
	Management of Descriptors	175
	Descriptor Locking	177
	Implementation of Locking	178
	Multiplexing I/O on Descriptors	179
	Implementation of Select	181
	Moving Data Inside the Kernel	184
	Exercises	185
	References	186

Chapter 7 The Filesystem

187

7.1	Structure and Overview	187
	Directories	187
	Links	189
	Quotas	191
7.2	Overview of the Internal Filesystem	191
	Allocating and Finding the Blocks on the Disk	193
7.3	Internal Structure and Redesign	195
	New Filesystem Organization	196
	Optimizing Storage Utilization	198

Filesystem Parameterization	200
Layout Policies	201
7.4 Filesystem Data Structures	203
Inode Management	205
Finding File Blocks	206
File-Block Allocation	207
7.5 Buffer Management	208
Implementation of Buffer Management	211
7.6 Quotas	213
7.7 Allocation Mechanisms	217
7.8 Translation of Filesystem Names	219
Exercises	221
References	223

Chapter 8 Device Drivers

225

8.1 Overview	225
8.2 Device Drivers	227
I/O Queueing	228
Interrupt Handling	229
8.3 Block Devices	229
8.4 Character Devices	230
Raw Devices and Physical I/O	231
Character-Oriented Devices	233
Entry Points for Character Device Drivers	233
8.5 Autoconfiguration	234
Probing for Devices	235
Attaching a Device	236
Device Naming	236
8.6 UNIBUS Devices	237
The <i>up</i> Device Driver	238
Autoconfiguration Support	239
Logical-to-Device Mapping of I/O Requests	242
I/O Strategy	243
Disksort	244
Drive-Positioning Algorithm	245
Initiating an I/O Operation	246
Interrupt Handling	247
UNIBUS Adapter Support Routines	249
8.7 MASSBUS Devices	253
Autoconfiguration	253
I/O Strategy	253
Interrupt Handling	254
Exercises	256

Chapter 9 Terminal Handling

259

9.1 Terminal Processing Modes	259
9.2 Line Disciplines	260

9.3	User Interface	262
9.4	The <i>tty</i> Structure	263
9.5	Process Groups and Terminal Control	265
9.6	C-lists	266
9.7	RS-232 and Modem Control	267
9.8	Terminal Operations	268
	Open	268
	Output Line Discipline	268
	Output Top Half	270
	Output Bottom Half	271
	Input Bottom Half	271
	Input Top Half	273
	The <i>stop</i> Routine	273
	The <i>ioctl</i> Routine	274
	Modem Transitions	275
	Closing Terminal Devices	275
9.9	Other Line Disciplines	276
	Berknet	276
	Serial Line IP Discipline	276
	Graphics Tablet Discipline	277
9.10	Summary	277
	Exercises	277
	References	278

Part 4 Interprocess Communication 279

Chapter 10 Interprocess Communication 281

10.1	Interprocess-Communication Model	282
	Using Sockets	284
10.2	Implementation Structure and Overview	288
10.3	Memory Management	289
	Mbufs	289
	Storage-Management Algorithms	291
	Mbuf Utility Routines	292
10.4	Data Structures	292
	Communication Domains	293
	Sockets	294
	Socket Addresses	296
10.5	Connection Setup	298
10.6	Data Transfer	300
	Transmitting Data	301
	Receiving Data	302
	Passing Access Rights	304
	Access Rights in the UNIX Domain	305
10.7	Socket Shutdown	306
	Exercises	307
	References	309

Chapter 11 Network Communication 311

- 11.1 Internal Structure 312
 - Data Flow 312
 - Communication Protocols 314
 - Network Interfaces 315
- 11.2 Socket-to-Protocol Interface 318
 - Protocol User-Request Routine 318
 - Internal Requests 321
 - Protocol Control-Output Routine 322
- 11.3 Protocol-Protocol Interface 322
 - pr_output* 323
 - pr_input* 323
 - pr_ctlinput* 323
- 11.4 Protocol-Network-Interface Interface 324
 - Packet Transmission 324
 - Packet Reception 325
- 11.5 Routing 327
 - Routing Tables 328
 - Routing Redirects 329
 - Routing-Table Interface 330
 - User-Level Routing Policies 330
- 11.6 Buffering and Congestion Control 331
 - Protocol Buffering Policies 331
 - Queue Limiting 332
- 11.7 Raw Sockets 332
 - Control Blocks 332
 - Input Processing 333
 - Output Processing 334
- 11.8 Additional Network Subsystem Topics 334
 - Out-of-Band Data 334
 - Address Resolution Protocol 335
 - VAX UNIBUS Interfaces 336
 - Trailer Protocols 338
 - Exercises 340
 - References 341

Chapter 12 Network Protocols 343

- 12.1 DARPA Internet Network Protocols 343
 - Internet Addresses 345
 - Subnets 346
 - Broadcast Addresses 347
 - Internet Ports and Associations 348
 - Protocol Control Blocks 348
- 12.2 User Datagram Protocol (UDP) 350
 - Initialization 350
 - Output 350

311

	Input	351
	Control Operations	352
12.3	Internet Protocol (IP)	352
	Output	353
	Input	354
	Forwarding	356
12.4	Transmission Control Protocol (TCP)	357
	TCP Connection States	358
	Sequence Variables	360
12.5	TCP Algorithms	362
	Timers	363
	Estimation of Round-Trip Time	365
	Connection Establishment	366
	Connection Shutdown	367
12.6	TCP Input Processing	368
12.7	TCP Output Processing	371
	Sending Data	371
	Avoidance of the Silly-Window Syndrome	372
	Avoidance of Small Packets	373
	Window Updates	374
	Retransmit State	375
	Source-Quench Processing and Congestion Control	375
	Slow Start	376
	Avoidance of Congestion with Slow Start	377
12.8	Internet Control Message Protocol (ICMP)	378
12.9	ARPANET Host Interface	380
12.10	Xerox Network Systems Communication Domain (XNS)	381
	XNS Control Operations	383
12.11	Summary	384
	Creating a Communication Channel	384
	Sending and Receiving Data	385
	Terminating Data Transmission and/or Reception	386
	Exercises	387
	References	389

Part 5 System Operation

391

343

Chapter 13 System Startup

393

13.1	Overview	393
13.2	Bootstrapping	394
	The boot Program	394
	VAX Console Media	396
13.3	Kernel Initialization	396
	Assembly-Language Startup	397
	Machine-Dependent Initialization	398
	Message Buffer	399

	System Data Structures	399	
	Memory Allocator	400	
	Autoconfiguration	400	
	Machine-Independent Initialization	403	
13.4	User-Level Initialization	405	
	/etc/init	406	
	/etc/rc	406	
	/etc/getty	407	
	/bin/login	407	
13.5	System Startup Topics	407	
	Kernel Configuration	408	
	System Shutdown and Autoreboot	409	
	System Debugging	410	
	Exercises	410	
	References	411	
	Glossary		413
	Index		451

PART 1

Overview

413

451

History and Goals

1.1 History of the UNIX System

The UNIX system is one of the older of the widely used operating systems, with a history of almost 20 years, but many of its most distinctive and useful facilities are relatively recent developments.

Origins

The first version of the UNIX system was developed at Bell Laboratories in 1969 by Ken Thompson as a private research project to use an otherwise idle PDP-7. Thompson also designed the language B, in which much of the early system was soon rewritten. He was joined shortly thereafter by Dennis Ritchie, who not only contributed to the design and implementation of the system, but also invented the C programming language in which later versions are written. The original elegant design of the system [Ritchie, 1978] and developments [Ritchie, 1984a] of the past 15 years [Compton, 1985] have made the UNIX system an important and powerful operating system [Ritchie, 1987].

Ritchie, Thompson, and other early Research UNIX developers had previously worked on the Multics project [Peirce, 1985; Organick, 1975], which had a strong influence on the newer operating system. Even the name "UNIX" is merely a pun on Multics; in areas where Multics attempted to do many things, UNIX tried to do one thing well. The basic organization of the UNIX filesystem, the idea of the command interpreter (the shell) being a user process, the general organization of the filesystem interface, and many other system characteristics, come directly from Multics.

Ideas from various other operating systems, such as the Massachusetts Institute of Technology's (MIT's) CTSS, also have been used. The *fork* operation to create new processes comes from Berkeley's GENIE (SDS-940, later XDS-940)

operating system. Allowing a user to create processes inexpensively led to using one process per command, rather than running commands as procedure calls, as is done in Multics.

There are at least three major streams of development of the UNIX system. Figure 1.1 sketches their evolution, especially of those branches leading to 4.3BSD and to System V [Chambers & Quarterman, 1983; Uniejewski, 1985]. The dates given are approximate, and there is no attempt to show all influences. Some of the systems named in the figure are not mentioned in the text, but are included to show more clearly the relations among the ones that we shall examine.

Research UNIX

The first major editions were the Research systems from Bell Laboratories. In addition to the earliest versions of the system, these include **UNIX Time-Sharing System, Sixth Edition**, commonly known as **V6**, which, in 1976, was the first version widely available outside Bell Laboratories. Systems are identified by the edition numbers of the UNIX Programmer's Manual that were current when the distributions were made.

The system was distinguished from other operating systems in three important ways:

1. The system was written in a high-level language
2. The system was distributed in source form
3. The system provided powerful primitives normally found in only those operating systems that ran on much more expensive hardware

Most of the system source was written in C rather than in assembly language. The prevailing belief at the time was that an operating system had to be written in assembly language to provide reasonable efficiency and to get access to the hardware. The C language itself was at just a high enough level to allow it to be easily compiled for a wide range of computer hardware, without its being so complex or restrictive that systems programmers had to revert to assembly language to get reasonable efficiency or functionality. Access to the hardware was provided through assembly-language stubs for the 3 percent of the operating-system functions—such as context switching—that needed them. Although the success of UNIX does not stem solely from the fact that it was written in a high-level language, the use of C was a critical first step [Ritchie *et al.*, 1978; Kernighan & Ritchie, 1978; Kernighan & Ritchie, 1988]. Ritchie's C language is descended [Rosler, 1984] from Thompson's B language, which was itself descended from BCPL [Richards & Whitby-Stevens, 1982]. C continues to evolve [Tuthill, 1985; X3J11, 1988], and there is a variant, C++, that more readily permits data abstraction [Stroustrup, 1984; USENIX, 1987].

The second important distinction of UNIX was its early release from Bell Laboratories to other research environments in source form. By providing source, the system's founders ensured that other organizations would be able not only to

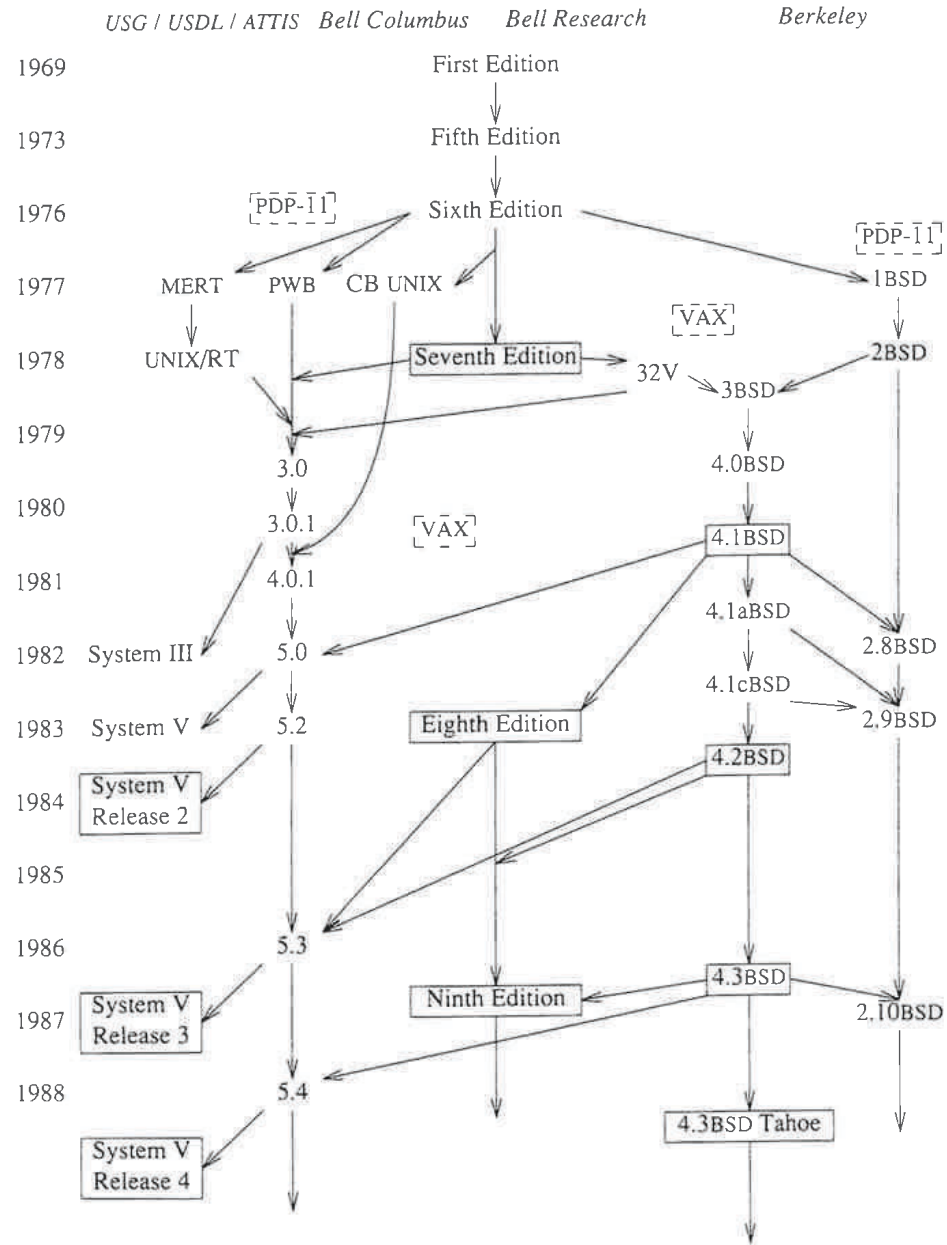


Figure 1.1 The UNIX system family tree.

use the system, but also to tinker with its inner workings. The ease with which new ideas could be adopted into the system always has been key to the changes that have been made to it. Whenever a new system that tried to upstage UNIX came along, somebody would dissect the newcomer and clone its central ideas into UNIX. The unique ability to use a small, comprehensible system, written in a

high-level language, in an environment swimming in new ideas led to a UNIX system that evolved far beyond its humble beginnings.

The third important distinction of UNIX was that it provided individual users with the ability to run multiple processes concurrently and to connect these processes into pipelines of commands. At the time, only operating systems running on large and expensive machines had the ability to run multiple processes, and the number of concurrent processes usually was tightly controlled by a system administrator.

Most early UNIX systems ran on the PDP-11, which was inexpensive and powerful for its time. Nonetheless, there was at least one early port of Sixth Edition UNIX to a machine with a rather different architecture [Miller, 1978]. The PDP-11 also had an inconveniently small address space. The introduction of machines with 32-bit address spaces, especially the VAX-11/780, provided an opportunity for UNIX to expand its services to include virtual memory and networking. Earlier experiments by the Research group in providing UNIX-like facilities on different hardware had led to the conclusion that it was as easy to move the entire operating system as it was to duplicate its services under another operating system. The first UNIX system with portability as a specific goal was **UNIX Time-Sharing System, Seventh Edition (V7)**, which ran on the PDP-11 and the Interdata 8/32, and had a VAX variety called **UNIX/32V Time-Sharing, System Version 1.0 (32V)**. The Research group at Bell Laboratories has also developed **UNIX Time-Sharing System, Eighth Edition (V8)**. Their current system is **UNIX Time-Sharing System, Ninth Edition (V9)**.

AT&T UNIX System III and System V

After the distribution of Seventh Edition in 1978, the Research group turned over external distributions to the UNIX Support Group (USG). USG had previously distributed such systems as the **UNIX Programmer's Work Bench (PWB)** internally, and sometimes externally as well [Mohr, 1985].

Their first external distribution after Seventh Edition was **UNIX System III (System III)**, in 1982, which incorporated features of Seventh Edition, of 32V, and also of several UNIX systems developed by groups other than the Research group. Features of UNIX/RT (a real-time UNIX system) were included, as well as many features from PWB. USG released **UNIX System V (System V)** in 1983; that system is largely derived from System III. The court-ordered divestiture of the Bell Operating Companies from AT&T has left AT&T in a position to market System V aggressively [Wilson, 1985; Bach, 1986].

USG metamorphosed into the UNIX System Development Laboratory (USDL), which released **UNIX System V, Release 2** in 1984. System V, Release 2, Version 4 introduced paging [Miller, 1984; Jung, 1985], including copy-on-write and shared memory, to System V. The System V implementation was not based on the Berkeley paging system. USDL was succeeded by AT&T Information Systems (ATTIS), which distributed **UNIX System V, Release 3** in 1987. That system included STREAMS, an IPC mechanism adopted from V8 [Presotto & Ritchie, 1985].

Other Organizations

The ease with which the UNIX system can be modified has led to development work at numerous organizations, including the Rand Corporation, responsible for the Rand ports mentioned in Chapter 10; Bolt Beranek and Newman (BBN), who produced the direct ancestor of the 4.2BSD networking implementation discussed in Chapter 12; the University of Illinois, which did earlier networking work; Harvard; Purdue; and Digital Equipment Corporation (DEC).

Probably the most widespread version of the UNIX operating system, according to the number of machines on which it runs, is XENIX by Microsoft Corporation. XENIX was based originally on Seventh Edition, but later on System V. More recently, Microsoft and AT&T have agreed to merge the two systems.

Berkeley Software Distributions

The most influential of the non-Bell Laboratories and non-AT&T UNIX development groups has been the University of California at Berkeley [McKusick, 1985]. UNIX software from Berkeley is released in **Berkeley Software Distributions** (BSD); for example, as 4.3BSD. The first Berkeley VAX UNIX work was the addition to 32V of virtual memory, demand paging, and page replacement in 1979 by William Joy and Özalp Babaoğlu to produce 3BSD [Babaoğlu & Joy, 1981]. The large virtual-memory space of 3BSD was motivated by the development of very large programs, such as Berkeley's own *Franz* LISP. This memory-management work convinced the Defense Advanced Research Projects Agency (DARPA) to fund the Berkeley team for the later development of a standard UNIX system (4BSD) for DARPA's contractors to use.

One of the goals of this project was to provide support for the DARPA Internet networking protocols, TCP/IP [Cerf & Cain, 1983]. The networking implementation was general enough to communicate among diverse network facilities, ranging from local networks, such as Ethernets and token rings, to long-haul networks, such as DARPA's ARPANET.

We refer to all the Berkeley VAX UNIX systems following 3BSD as 4BSD, although there were actually several releases—4.0BSD, 4.1BSD, 4.2BSD, 4.3BSD, and 4.3BSD Tahoe. 4BSD was the operating system of choice for VAXes from the time the VAX first became available in 1977 until the release of System III (1979 to 1982), and it remains so for many research or networked installations. Most organizations would buy a 32V license, but order 4BSD from Berkeley. Many installations inside the Bell System ran 4.1BSD (many now run 4.3BSD).

The 4BSD work for DARPA was guided by a steering committee that included many notable people from both commercial and academic institutions. The culmination of the original Berkeley DARPA UNIX project was the release of **4.2BSD** in 1983; further research at Berkeley has already produced **4.3BSD**—and work still proceeds.

UNIX in the World

Dozens of computer manufacturers,¹ including almost all the ones usually considered “major” by market share, have either announced or introduced computers that run the UNIX system or close derivatives, and numerous other companies sell related peripherals, software packages, support, training, documentation, or combinations of these. The hardware packages involved range from micros through minis, multis, and mainframes to supercomputers. Most of these use ports of System V, 4.2BSD, 4.3BSD, or mixtures, although there is still a variety of machines running software based on System III, 4.1BSD, and Seventh Edition. There are PDP-11s running 2BSD and other UNIX variants. There are even some Sixth Edition systems still in regular operation.

The UNIX system is also a fertile field for academic endeavor. Thompson and Ritchie were given the Association for Computing Machinery Turing award for the design of the system [Ritchie, 1984b]. It and related, specially designed teaching systems—such as Tunis [Ewens *et al.*, 1985; Holt, 1983], XINU [Comer, 1984], and MINIX [Tanenbaum, 1987]—are widely used in courses on operating systems. The UNIX system is ubiquitous in universities and research facilities throughout the world, and is ever more widely used in industry and commerce.

1.2 BSD and Other Systems

The Computer Systems Research Group (CSRG) incorporated features not only from UNIX systems, but also from other operating systems. Many of the features of the 4BSD terminal drivers are from TENEX/TOPS-20. Job control (in concept, not implementation) is derived from that of TOPS-20 and from the MIT Incompatible Timesharing System (ITS). The virtual-memory interface first proposed for 4.2BSD, and since implemented by several commercial vendors, was based on the file-mapping and page-level interfaces that first appeared in TENEX/TOPS-20. Multics has often been a reference point in the design of new facilities.

The quest for efficiency has been a major factor in much CSRG work. Some efficiency improvements have been made due to comparisons with the proprietary operating system for the VAX, VMS [Kashtan, 1980;].

Other UNIX variants have adopted several 4BSD features. Both AT&T UNIX System V [AT&T, 1987] and the IEEE 1003.1 POSIX standard [P1003.1, 1988] have adopted these filesystem system calls (see Chapter 6):

¹ These manufacturers include Altos, Alliant, Amdahl, Apollo, Apple, Ardent Computer, AT&T, Bull, Callan, Celerity, Codata, Convergent Technologies, Convex, COSI, Cray, Cromemco, Data General, DEC, Dual Systems, ELXSI, Encore, Ericsson, Fairchild, Flexible, Gould, Heurikon, Hewlett-Packard, Honeywell, IBM, ICON, ICL, Integrated Business Computers, Integrated Solutions, Intel, Interactive Systems, Locus, Logical MicroComputer, Medical Informatics, Microsoft, MIPS, NBI, NCR, National Semiconductor, NeXT, Nixdorf, Olivetti, Onyx, Pacific Computer, Parallel, Perkin-Elmer, Philips, Plexus, Prime, Pyramid, R Systems, Ridge, Sequent, Siemens, Silicon Graphics, Sony, Stellar, Sun Microsystems, Symmetric, Tandy, Tektronix, Unisys, Visual Technology, and WICAT.

- *rename* for renaming files and directories
- *mkdir* and *rmdir* for directory creation and deletion

Both have also adopted the directory-access routines.

In addition, POSIX and the related National Bureau of Standards (NBS) Federal Information Processing Standard (FIPS) have adopted

- Reliable signals (Chapter 4)
- Job control (Chapter 2)
- Multiple file-access permission groups (Chapter 6)

The X/OPEN Group, originally solely European vendors, but now including several U.S. companies, produced the X/OPEN Portability Guide [X/OPEN, 1987], a document that specifies both the kernel interface and many of the utility programs available to UNIX system users. X/OPEN has adopted many of the POSIX facilities. Other, similar, standards and guides are expected to adopt them, as well. The IEEE 1003.1 standard is also an ISO Draft International Standard, named SC22 WG15. Thus, the POSIX facilities will probably be accepted in most UNIX-like systems worldwide.

The 4BSD *socket* interprocess-communication mechanism (see Chapter 10) was designed for portability and was immediately ported to AT&T System III, although it was never distributed with that system. The 4BSD implementation of the TCP/IP networking protocol suite (see Chapter 12) is widely used as the basis for further implementations on systems ranging from AT&T 3B machines running System V to VMS to IBM PCs.

CSRG also continues to cooperate closely with vendors whose systems are based on 4.2BSD and 4.3BSD. This simultaneous development contributes to the ease of further ports of 4.3BSD, and to ongoing development of the system.

The Influence of the User Community

Much of the Berkeley UNIX development work was done in response to the user community. Ideas and expectations came not only from DARPA, the principal direct-funding organization, but also from many of the users of the system at companies and universities worldwide.

The Berkeley researchers accepted not only ideas from the user community, but also actual software. Contributions to 4BSD came from numerous universities and other organizations in Australia, Canada, Europe, and the United States. These contributions included major features, such as autoconfiguration and disk quotas. A few ideas, such as the *fcntl* system call, were taken from System V, although licensing and pricing considerations have prevented the use of any actual code from System III or System V in 4BSD. In addition to contributions that are included in the distributions proper, CSRG also distributes a set of user-contributed software.

An example of a community-developed facility is the public-domain time-zone-handling package that was adopted with the 4.3BSD Tahoe release. It was designed and implemented by an international group, including Arthur Olson, Robert Elz, and Guy Harris, partly due to discussions in the USENET newsgroup **comp.std.unix**. This package takes time-zone-conversion rules completely out of the C library, putting them in files that require no system-code changes in order to change time-zone rules; this change is especially useful with binary-only distributions of UNIX. The method also allows individual processes to choose rules, rather than keeping one ruleset specification systemwide. The distribution includes a large database of rules used in many areas throughout the world, from China to Australia to Europe. Distributions of the UNIX system are thus simplified, because it is not necessary to have the software set up differently for different destinations, as long as the whole database is included.

Berkeley accepts mail about bugs and their fixes at a well-known electronic address, **4bsd-bugs@berkeley.edu**, and the UNIX software house MT XINU distributes a bug list compiled from such submissions. Many of the bug fixes are incorporated in future distributions. There is constant discussion of UNIX in general (including 4.3BSD) in the DARPA Internet mailing list UNIX-WIZARDS, which appears on the USENET network as the newsgroup **comp.unix.wizards**; both the Internet and USENET are international in scope. There is another USENET newsgroup dedicated to 4BSD bugs: **comp.bugs.4bsd**. Few ideas have been accepted by Berkeley directly from these lists and newsgroups, because of the difficulty of sifting through the voluminous submissions. But there is now a moderated newsgroup dedicated to CSRG-sanctioned fixes to such bugs, called **comp.bugs.4bsd.bug-fixes**, and discussions in these newsgroups sometimes lead to new facilities being written that are later accepted.

1.3 Design Goals of 4BSD

4BSD is a research system developed for and partly by a research community, and, more recently, a commercial community. The developers considered many design issues as they wrote the system. There were nontraditional considerations in and inputs into the design, which nevertheless yielded results with commercial importance.

The early systems were technology-driven. They took advantage of current hardware that was unavailable in other UNIX systems. This new technology included

- Virtual-memory support
- Device drivers for third-party (non-DEC) peripherals
- Terminal-independent support libraries for screen-based applications; numerous applications were developed that used these libraries, including the screen-based editor *vi*.

4BSD's support of a huge number of popular third-party peripherals, compared to the AT&T distribution's meager offerings in 32V, was an important factor in 4BSD popularity. Until other vendors began providing their own support of 4.2BSD-based systems, there was no alternative for universities that had to minimize hardware costs.

Terminal-independent screen support, although it may now seem rather pedestrian, was at the time very important in Berkeley software's popularity.

4.2BSD Design Goals

DARPA wanted Berkeley to develop 4.2BSD as a standard research operating system for the VAX. Many new facilities were designed for inclusion in 4.2BSD. These facilities included a completely revised virtual-memory system to support processes with large sparse address space, a much higher-speed filesystem, interprocess communication facilities, and networking support. The high-speed filesystem and revised virtual-memory system were needed by researchers doing computer-aided design and manufacturing (CAD/CAM), image processing, and artificial intelligence (AI). The interprocess communication facilities were needed by sites doing research in distributed systems. The networking support was primarily motivated by DARPA's interest in connecting their researchers through the 56-Kbit per second ARPA Internet (although Berkeley was also interested in getting good performance over higher-speed local-area networks).

No attempt was made to provide a true distributed operating system [Popek, 1981]. Instead, the traditional ARPANET goal of resource sharing was used. There were three reasons that a resource-sharing design was chosen:

- The systems were widely distributed and demanded administrative autonomy. At the time, a true distributed operating system required a central administrative authority.
- The known algorithms for tightly coupled systems did not scale well.
- Berkeley's charter was to incorporate current, proven software technology, rather than to develop new, unproven technology.

Therefore, easy means were provided for remote login (*rlogin*, *telnet*), file transfer (*rcp*, *ftp*), and remote command execution (*rsh*), but all host machines retained separate identities that were not hidden from the users.

Due to time constraints, the system that was released as 4.2BSD did not include all the facilities that were originally intended to be included. In particular, the revised virtual-memory system was not part of the 4.2BSD release. CSRG did, however, continue its ongoing effort to track fast-developing hardware technology in several areas. The networking system supported a wide range of hardware devices, including multiple interfaces to 10-Mbit/s Ethernet, to ring networks, and to NSC's Hyperchannel. The kernel sources were modularized and rearranged to ease portability to new architectures, including to microprocessors and to larger machines.

4.3BSD Design Goals

Problems with 4.2BSD were among the motivations for 4.3BSD. Because 4.2BSD included many new facilities, there were bugs in some of them, particularly in the TCP protocol implementation. There was a loss of performance, partly due to the introduction of symbolic links. Some facilities were not included due to lack of time. Others, such as TCP/IP subnet and routing support, were not specified soon enough by outside parties for them to be incorporated in the 4.2BSD release.

Commercial systems usually maintain backward compatibility for many releases so as not to make existing applications obsolete. Maintaining compatibility is increasingly difficult, however, so most research systems maintain little or no backward compatibility. As a compromise to other researchers, the BSD releases are usually backward compatible for one release, but have the deprecated facilities clearly marked. This approach allows for an orderly transition to the new interfaces without constraining the system from evolving smoothly. In particular, backward compatibility of 4.3BSD with 4.2BSD was considered highly desirable for application portability.

The C language interface to 4.3BSD differs from that of 4.2BSD in only a few commands to the terminal interface and in the use of one argument to one IPC system call (*select*: see Section 6.2). A flag was added in 4.3BSD to the system call that establishes a signal handler to allow a process to request the 4.1BSD semantics for signals, rather than the 4.2BSD semantics (see Section 4.7). The sole purpose of the flag was to allow existing applications that depended on the old semantics to continue working without being rewritten.

The implementation changes between 4.2BSD and 4.3BSD often are not visible to users, but they are numerous. For example, changes were made to improve support for multiple network-protocol families, such as XEROX NS, in addition to TCP/IP. Since 4.3BSD retains the advances of the earlier Berkeley systems, this book's examination of 4.3BSD is an investigation of the major features of the Berkeley systems.

The second release of 4.3BSD, hereafter referred to as 4.3BSD Tahoe, added support for the Computer Consoles, Inc. (CCI) Power 6 (Tahoe) series of mini-computers in addition to the VAX. Although generally similar to the original release of 4.3BSD for the VAX, it includes many modifications and new features.

Future Berkeley Releases

4.3BSD is not perfect. In particular, the virtual-memory system needs to be completely replaced. The new virtual-memory system needs to provide algorithms that are better suited to the large memories and slow disks currently available, and needs to be less VAX architecture dependent. The terminal driver has been carefully kept compatible not only with Seventh Edition, but even with Sixth Edition. This feature has been useful, but is less so now, especially considering the lack of orthogonality of its commands and options. CSRG plans to distribute a POSIX-compatible terminal driver; since System V will be compliant with POSIX, the

terminal driver will also be compatible with System V. POSIX compatibility in general is a goal.

Other currently planned BSD work includes development of an implementation of networking protocols in the International Organization for Standardization (ISO) suite, further TCP/IP performance improvements and enhancements, and the implementation of other networking protocols.

The most critical shortcoming of 4.3BSD is the lack of a distributed filesystem. As with networking protocols, there is no single distributed filesystem that provides enough speed and functionality for all situations. It is frequently necessary to support several different distributed filesystem protocols, just as it is necessary to run several different network protocols. Thus, a standard interface to filesystems somewhat more general than but very similar to Sun Microsystems' framework for their Network File System (NFS) is being developed so that multiple local and remote filesystems can be supported, much as multiple networking protocols are supported by 4.3BSD [Sandberg, 1985].

Original work on the flexible configuration of IPC processing modules was done at Bell Laboratories in UNIX Eighth Edition [Presotto & Ritchie, 1985]. This *stream I/O system* was based on the UNIX character I/O system. It allowed a user process to open a raw terminal port and then to insert appropriate kernel-processing modules, such as one to do normal terminal line editing. Modules to process network protocols also could be inserted. Stacking a terminal-processing module on top of a network-processing module allowed flexible and efficient implementation of *network virtual terminals* within the kernel. A problem with stream modules, however, is that they are inherently linear in nature, and thus they do not adequately handle the fan-in and fan-out associated with multiplexing in datagram-based networks; such multiplexing is done in device drivers, below the modules proper. The Eighth Edition stream I/O system was adopted in System V, Release 3 as the STREAMS system.

The design of the networking facilities for 4.2BSD took a different approach, based on the *socket* interface and a flexible multilayer network architecture. This design allows a single system to support multiple sets of networking protocols with stream, datagram, and other types of access. Protocol modules may deal with multiplexing of data from different connections onto a single transport medium, as well as with demultiplexing of data for different protocols and connections received from each network device.

There are plans to redesign the internal protocol layering using ideas from the V8 stream I/O system. A socket interface will be used, rather than a character-device interface, and demultiplexing will be handled internally by the network protocols in the kernel. However, like streams, the interfaces between kernel protocol modules above the multiplexed layers will follow a uniform convention. This convention will allow the incorporation of terminal-processing modules into a network stream, producing efficient network virtual-terminal connections. It will also allow kernel support for remote-procedure protocols based on standard transport protocols. Finally, this interface will provide a mechanism to extend the kernel protocol framework into user processes, to allow prototyping of new protocols and to do network-monitoring functions.

1.4 Release Engineering

CSRG has always been a small group of software developers. This resource limitation requires careful software-engineering management. Careful coordination is needed not only of CSRG personnel, but also of members of the general community who contribute to the development of the system. Major distributions usually alternate between

- Major new facilities: 3BSD, 4.0BSD, 4.2BSD
- Bug fixes and efficiency improvements: 4.1BSD, 4.3BSD

This alternation allows release in a timely manner, while providing for refinement and correction of the new facilities and the elimination of performance problems produced by the new facilities. The timely followup of releases that include new facilities reflects the importance CSRG places on providing a reliable and robust system on which its user community can depend.

Developments from CSRG are released in three steps: alpha, beta, and final, as shown in Table 1.1. Alpha and beta releases are not true distributions—they are test systems. Alpha releases are normally available to only a few sites, most of those within the University. More sites get beta releases, but they do not get them directly; a tree structure is imposed to allow bug reports, fixes, and new software to be collected, evaluated, and checked for redundancies by first-level sites before forwarding to CSRG. For example, 4.1aBSD ran at more than 100 sites, but there were only about 15 primary beta sites. The beta-test tree allowed the developers at CSRG to concentrate on actual development rather than sifting through details from every beta-test site. This book was reviewed for technical accuracy by a similar process.

Many of the primary beta-test personnel not only had copies of the release running on their own machines, but also had login accounts on the development machine at Berkeley. Such users were commonly found logged in at Berkeley over the ARPA Internet, or sometimes via telephone dialup, from places far away, such as Massachusetts, Utah, Maryland, Texas, and Illinois, and from closer places, such as Stanford. For the 4.3BSD release, certain accounts and users had permission to modify the master copy of the system source directly. A number of

Table 1.1 Release test steps.

Description	Release Steps			
	alpha	internal	beta	final
name:	4.1aBSD	4.1bBSD	4.1cBSD	4.2BSD
major new facility:	networking	fast filesystem	IPC	revised signals

limitation is unusually

element problems are new robust

final, —they, most not get d new t-level in 100 allowed sifting technical

release ment Berkeley away, closer rs had ber of

facilities, such as the Fortran and C compilers, as well as important system programs, such as *telnet* and *ftp*, include significant contributions from people who did not work for CSRG. One important exception to this approach was that changes to the kernel were made by only CSRG personnel, although the changes often were suggested by the larger community.

People given access to the master sources were carefully screened beforehand, but were not closely supervised. Their work was checked at the end of the beta-test period by CSRG personnel, who did a complete comparison of the source of the previous release with the current master sources—for example, of 4.3BSD with 4.2BSD. Facilities deemed inappropriate, such as new options to the directory-listing command or a changed return value for the *fseek()* library routine, were removed from the source before final distribution.

4BSD releases have usually included a pair of documents detailing changes to every user-level command [McKusick *et al.*, 1986] and to every kernel source file [Karels, 1986]. These documents are delivered with the final distribution. A user can look up any command by name and see immediately what has changed, and a developer can similarly look up any kernel file by name and get a summary of that file's changes.

This process illustrates an advantage of having a small number of principal developers: The developers all know the whole system thoroughly enough to be able to coordinate their own work with that of other people to produce a coherent final system. Companies with large development organizations find this result difficult to duplicate.

There is no CSRG marketing division. Thus, technical decisions are made largely for technical reasons, and are not driven by marketing promises. The Berkeley developers have been fanatical about this, and are well known for never promising delivery on a specific date.

References

AT&T, 1987.

AT&T, *The System V Interface Definition (SVID)*, Issue 2, American Telephone and Telegraph, Murray Hill, NJ (January 1987).

Babaoğlu & Joy, 1981.

Ö. Babaoğlu & W. N. Joy, "Converting a Swap-Based System to Do Paging in an Architecture Lacking Page-Referenced Bits," *Proceedings of the Eighth Symposium on Operating Systems Principles*, pp. 78–86 (December 1981).

Bach, 1986.

M. J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, Englewood Cliffs, NJ (1986).

Cerf & Cain, 1983.

V. G. Cerf & E. Cain, *The DoD Internet Architecture Model*, Elsevier Science, Amsterdam, Netherlands (1983).

Chambers & Quarterman, 1983.

J. B. Chambers & J. S. Quarterman, "UNIX System V and 4.1C BSD," *USENIX Association Conference Proceedings*, pp. 267–291 (June 1983).

Comer, 1984.

D. Comer, *Operating System Design: The Xinu Approach*, Prentice-Hall, Englewood Cliffs, NJ (1984).

Compton, 1985.

M. Compton, editor, "The Evolution of UNIX," *UNIX Review* 3(1) (January 1985).

Ewens *et al.*, 1985.

P. Ewens, D. R. Blythe, M. Funkenhauser, & R. C. Holt, "Tunis: A Distributed Multiprocessor Operating System," *USENIX Association Conference Proceedings*, pp. 247–254 (June 1985).

Holt, 1983.

R. C. Holt, *Concurrent Euclid, the UNIX System, and Tunis*, Addison-Wesley, Reading, MA (1983).

Joy, 1980.

W. N. Joy, "Comments on the Performance of UNIX on the VAX," Technical Report, University of California Computer System Research Group, Berkeley, CA (April 1980).

Jung, 1985.

R. S. Jung, "Porting the AT&T Demand Paged UNIX Implementation to Microcomputers," *USENIX Association Conference Proceedings*, pp. 361–370 (June 1985).

Karels, 1986.

M. J. Karels, "Changes to the Kernel in 4.3BSD," pp. 13:1–32 in *UNIX System Manager's Manual, 4.3 Berkeley Software Distribution, Virtual VAX-11 Version*, USENIX Association, Berkeley, CA (1986).

Kashtan, 1980.

D. L. Kashtan, "UNIX and VMS: Some Performance Comparisons," Technical Report, SRI International, Menlo Park, CA (February 1980).

Kernighan & Ritchie, 1978.

B. W. Kernighan & D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ (1978).

Kernighan & Ritchie, 1988.

B. W. Kernighan & D. M. Ritchie, *The C Programming Language*, 2nd ed, Prentice-Hall, Englewood Cliffs, NJ (1988).

McKusick, 1985.

M. K. McKusick, "A Berkeley Odyssey," *UNIX Review* 3(1), p. 30 (January 1985).

McKusick *et al.*, 1986.

M. K. McKusick, J. M. Bloom, & M. J. Karels, "Bug Fixes and Changes in 4.3BSD," pp. 12:1–22 in *UNIX System Manager's Manual, 4.3 Berkeley Software Distribution, Virtual VAX-11 Version*, USENIX Association, Berkeley, CA (1986).

BSD,"
33).

de-Hall,

January

. Distri-
ference

ddison-

VAX,"
research

ation to
35, pp.

n UNIX
Virtual

isions,"
30).

nguage,

2nd ed,

) (Janu-

anges in
Berkeley
ociation,

Miller, 1978.

R. Miller, "UNIX—A Portable Operating System," *ACM Operating System Review* **12**(3), pp. 32–37 (July 1978).

Miller, 1984.

R. Miller, "A Demand Paging Virtual Memory Manager for System V," *USENIX Association Conference Proceedings*, pp. 178–182 (June 1984).

Mohr, 1985.

A. Mohr, "The Genesis Story," *UNIX Review* **3**(1), p. 18 (January 1985).

Organick, 1975.

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

P1003.1, 1988.

P1003.1, *IEEE P1003.1 Portable Operating System Interface for Computer Environments (POSIX)*, Institute of Electrical and Electronic Engineers, Piscataway, NJ (1988).

Peirce, 1985.

N. Peirce, "Putting UNIX In Perspective: An Interview with Victor Vyssotsky," *UNIX Review* **3**(1), p. 58 (January 1985).

Popek, 1981.

B. Popek, "Locus: A Network Transparent, High Reliability Distributed System," *Proceedings of the Eighth Symposium on Operating Systems Principles*, pp. 169–177 (December 1981).

Presotto & Ritchie, 1985.

D. L. Presotto & D. M. Ritchie, "Interprocess Communication in the Eighth Edition UNIX System," *USENIX Association Conference Proceedings*, pp. 309–316 (June 1985).

Richards & Whitby-Strevens, 1982.

M. Richards & C. Whitby-Strevens, *BCPL: The Language and Its Compiler*, Cambridge University Press, Cambridge, UK (1980, 1982).

Ritchie, 1978.

D. M. Ritchie, "A Retrospective," *Bell System Technical Journal* **57**(6), pp. 1947–1969 (July–August 1978).

Ritchie *et al.*, 1978.

D. M. Ritchie, S. C. Johnson, M. E. Lesk, & B. W. Kernighan, "The C Programming Language," *Bell System Technical Journal* **57**(6), pp. 1991–2019 (July–August 1978).

Ritchie, 1984a.

D. M. Ritchie, "The Evolution of the UNIX Time-Sharing System," *AT&T Bell Laboratories Technical Journal* **63**(8), pp. 1577–1593 (October 1984).

Ritchie, 1984b.

D. M. Ritchie, "Reflections on Software Research," *Comm ACM* **27**(8), pp. 758–760 (1984).

Ritchie, 1987.

D. M. Ritchie, "Unix: A Dialectic," *USENIX Association Conference Proceedings*, pp. 29–34 (January 1987).

Rosler, 1984.

L. Rosler, "The Evolution of C—Past and Future," *AT&T Bell Laboratories Technical Journal* **63**(8), pp. 1685–1699 (October 1984).

Sandberg, 1985.

R. Sandberg, "The Design and Implementation of the Sun Network File System," *USENIX Association Conference Proceedings*, pp. 119–130 (June 1985).

Stroustrup, 1984.

B. Stroustrup, "Data Abstraction in C," *AT&T Bell Laboratories Technical Journal* **63**(8), pp. 1701–1732 (October 1984).

Tanenbaum, 1987.

A. S. Tanenbaum, *Operating Systems: Design and Implementation*, Prentice-Hall, Englewood Cliffs, NJ (1987).

Tuthill, 1985.

B. Tuthill, "The Evolution of C: Heresy and Prophecy," *UNIX Review* **3**(1), p. 80 (January 1985).

USENIX, 1987.

USENIX, *Proceedings of the C++ Workshop*, USENIX Association, Berkeley, CA (November 1987).

Uniejewski, 1985.

J. Uniejewski, *UNIX System V and BSD4.2 Compatibility Study*, Apollo Computer, Chelmsford, MA (March 1985).

Wilson, 1985.

O. Wilson, "The Business Evolution of the UNIX System," *UNIX Review* **3**(1), p. 46 (January 1985).

X/OPEN, 1987.

X/OPEN, *The X/OPEN Portability Guide (XPG)*, Issue 2, Elsevier Science, Amsterdam, Netherlands (1987).

X3J11, 1988.

X3J11, *X3.159 Programming Language C Standard*, Global Press, Santa Ana, CA (1988).

CHAPTER 2

Design Overview of 4.3BSD

2.1 UNIX Facilities and the Kernel

The UNIX kernel provides four basic facilities: processes, a filesystem, communications, and system startup. This section outlines where each of these four basic services is described in this book.

1. Processes constitute a thread of control in an address space. Mechanisms for creating, terminating, and otherwise controlling processes are described in Chapter 4. The system multiplexes separate virtual-address spaces for each process; this memory management is discussed in Chapter 5.
2. The user interface to the filesystem and devices is similar; common aspects are discussed in Chapter 6. The filesystem is a set of named files, organized in a tree-structured hierarchy of directories, and of operations to manipulate them, as presented in Chapter 7. Files reside on physical media such as disks, and there are device drivers to access them, as set forth in Chapter 8. Terminals are used to access the system; their device drivers are the subject of Chapter 9.
3. Communication mechanisms provided by traditional UNIX systems include device I/O (Chapters 8 and 9), simplex reliable byte streams between related processes (see pipes, Section 10.1), and notification of exceptional events (see signals, Section 4.7). 4.3BSD also has a general interprocess-communication facility; this is described in Chapter 10. This facility uses access mechanisms distinct from those of the filesystem, but, once a connection is set up, it is possible to access it as though it were a pipe. There is a general networking framework, discussed in Chapter 11, that is normally used as a layer underlying the IPC facility. Chapter 12 describes a particular networking implementation in detail.

4. Any real operating system has operational issues, such as how to start it running, as described in Chapter 13.

Sections 2.4 through 2.13 present introductory material related to Chapters 4 through 13. We shall define terms, mention basic system calls, and explore historical developments. Finally, we shall give the reasons for many major design decisions.

The Kernel

The *kernel* is the part of the system that provides the basic system facilities; it creates and manages processes, and provides functions to access the filesystem and communication facilities. These functions, called *system calls*, appear to user processes as library subroutines like any others. These system calls are the only interface processes have to these facilities. Details of the system-call mechanism are given in Chapter 3, as are descriptions of some kernel mechanisms that do not execute as the direct result of a process doing a system call.

Users ordinarily interact with the system through a command-language interpreter, called a *shell*, and perhaps through additional user application programs. Such programs and the shell are implemented as sets of processes. Details of such programs are beyond the scope of this book, which instead concentrates almost exclusively on the kernel.

A *kernel* in traditional operating-system terminology is a small nucleus of software that provides only the minimal facilities necessary for implementing additional operating-system services. In contemporary research operating systems such as Accent [Fitzgerald & Rashid, 1986], Tunis [Ewens *et al.*, 1985], and the V Kernel [Cheriton, 1988], this division of functionality is more than just a logical one. Services such as filesystems and networking protocols are implemented as client application processes of the nucleus or kernel.

The 4.3BSD kernel is not partitioned into multiple processes. This was a basic design decision in the earliest versions of UNIX. The first two implementations by Ken Thompson had no memory mapping at all, and thus made no hardware-enforced distinction between user and kernel space [Ritchie, 1988]. A message-passing system could have been implemented as readily as the actually implemented model of kernel and user processes. The latter was chosen for simplicity. And the early kernels were small. It has been largely the introduction of more and larger facilities (such as networking) into the kernel that has made their separation into user processes an attractive prospect—one that is being pursued in, for example, Mach [Accetta *et al.*, 1986].

The following sections describe the services provided by the 4.3BSD kernel, and give an overview of the services design. Later chapters describe the detailed design and implementation of these services as they appear in 4.3BSD.

2.2 Kernel Organization

In this section, we view the organization of the UNIX kernel in two ways:

it run-

pters 4
histor-
n deci-

ities; it
system
to user
ie only
hanism
do not

e inter-
grams.
of such
almost

leus of
renting
systems
d the V
logical
ted as

was a
menta-
ade no
38]. A
ctually
or sim-
tion of
le their
ued in,

kernel,
etailed

- As a static body of software, categorized by the functionality offered by the modules that make up the kernel
- By its dynamic operation, categorized according to the services provided to users

The largest part of the kernel implements the system services that applications access through system calls. In 4.3BSD, this software has been organized according to

- Basic kernel facilities: timer and system clock handling, descriptor management, and process management
- Memory-management support: paging and swapping
- Generic system interfaces: the I/O, control, and multiplexing operations performed on descriptors
- The filesystem: files, directories, pathname translation, file locking, and I/O buffer management
- Terminal-handling support: the terminal-interface driver and terminal line disciplines
- Interprocess-communication facilities: sockets
- Support for network communication: communication protocols and generic network facilities, such as routing

Most of the software in these categories is machine independent and is portable across different hardware architectures. One major exception is the memory-management support that was originally written for the VAX. Although it can be ported to other architectures (basically by emulating VAX memory management), rewriting it for a target architecture is generally a better idea. This has been done numerous times by vendors who have ported the system to non-VAX architectures.

The machine-dependent aspects of the kernel are isolated from the mainstream code. The software that is machine dependent includes

- Lowest-level system-startup actions
- Trap and fault handling
- Low-level manipulation of the run-time context of a process
- Configuration and initialization of hardware devices
- Run-time support for I/O devices

Table 2.1 summarizes the software that constitutes the 4.3BSD kernel for the VAX. The numbers in column 2 are for lines of C source code, header files, and

Table 2.1 Machine-independent software in the 4.3BSD kernel.

Category	Lines of Code	Percentage
headers	6,240	5.4
initialization	580	0.5
kernel facilities	5,440	4.7
generic interfaces	1,340	1.2
filesystem	6,650	5.7
filesystem disk quotas	1,430	1.2
interprocess communication	3,500	3.0
network communication	13,440	11.5
terminal handling	3,410	2.9
virtual memory	4,990	4.3
other machine independent	1,250	1.1
total machine independent	48,270	41.5

assembly language. Virtually all the software in the kernel is written in the C programming language; less than 3 percent is written in assembly language. As the statistics in Table 2.2 show, the machine-dependent software, excluding device support, accounts for a minuscule 4.3 percent of the kernel.

Only a small part of the kernel is devoted to initializing the system. This code is used when the system is *bootstrapped* into operation and is responsible for setting up the kernel hardware and software environment (see Chapter 13). Some operating systems (especially those with limited physical memory) discard or *overlay* the software that performs these functions after they have been executed. The 4.3BSD kernel does not reclaim the memory used by the startup code because

Table 2.2 VAX-dependent software in the 4.3BSD kernel.

Category	Lines of Code	Percentage
input/output device headers	5,725	4.9
input/output device support	36,800	31.6
network device headers	2,450	2.1
network device support	14,800	12.7
routines in assembly language	3,400	2.9
other machine dependent	5,025	4.3
total machine dependent	68,200	58.5

that memory space is only a small fraction of the overall resources available on a typical machine. Also, the startup code does not appear in one place in the kernel—it is scattered throughout, and it usually appears in places logically associated with what is being initialized. DEC's Ultrix, a 4.2BSD-derived system, does recover this code, however.

2.3 Kernel Services

The boundary between the kernel- and user-level code is enforced by hardware-protection facilities provided by the underlying hardware. The kernel operates in a separate address space that is inaccessible to user processes. Privileged operations such as starting I/O and halting the central processing unit (CPU) are available to only the kernel. Applications request services from the kernel with *system calls*. System calls are used to cause the kernel to execute complicated operations, such as writing data to secondary storage, and simple operations, such as returning the current time of day. All system calls appear *synchronous* to applications: While the kernel does the actions associated with a system call, the application is not permitted to run.

A system call usually is implemented as a hardware trap that changes the CPU's execution mode and the current address-space mapping. Parameters supplied by users in system calls are validated by the kernel before they are used to ensure the integrity of the system. All parameters passed into the kernel are copied into the kernel's address space. Parameters are copied to ensure that validated parameters are not changed as a side effect of the system call. System-call results are returned by the kernel either in hardware registers or by their values being copied into user-supplied buffers. Like parameters passed into the kernel, user buffers used for the return of results must also be validated to ensure that they are part of an application's address space. If the kernel encounters an error while processing a system call, it returns an error code to the user. For the C programming language, this error code is stored in the global variable *errno*, and the function call used to start the system call returns the value *-1*.

User applications and the kernel operate independently of each other. UNIX does not infringe on an application by placing I/O control blocks or other operating-system-related data structures in the application's address space. Each user-level application is provided an independent address space in which it executes. The kernel makes most state changes, such as suspending a process while another is running, invisible to the processes involved.

2.4 Process Management

UNIX supports a multitasking environment. Each task or thread of execution is termed a *process*. The *context* of a UNIX process consists of user-level state, including the contents of its address space and the run-time environment, and

kernel-level state, which includes scheduling parameters, resource controls, and identification information. The context includes everything used by the kernel in providing services for the process. Users can create processes, control the processes' execution, and receive notification when the processes' execution status changes. Every process is assigned a unique value, termed a *process identifier* (PID). This value is used by the kernel to identify a process when reporting status changes to a user, and by a user when referencing a process in a system call.

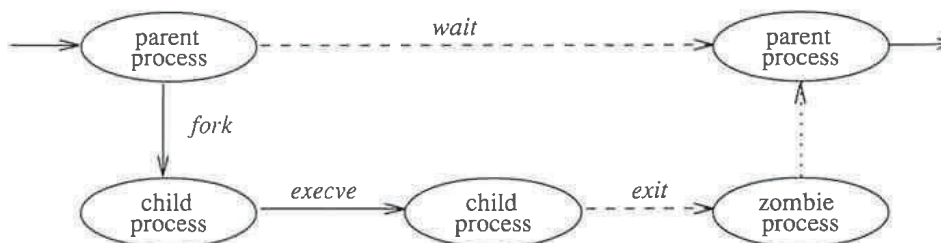
The kernel creates a process by duplicating the context of another process. The new process is termed a *child process* of the original *parent process*. The context duplicated in process creation includes both the user-level execution state of the process and the process's system state managed by the kernel. Important components of the kernel state are described in Chapter 4.

The process lifecycle is depicted in Fig. 2.1. A process may create a new process that is a copy of the original by using the *fork* system call. The *fork* call returns twice, once in the parent process, where the return value is the process identifier of the child, and once in the child process, where the return value is 0. The parent-child relationship induces a hierarchical structure on the set of processes in the system. The new process shares all its parent's resources, such as file descriptors, signal-handling status, and memory layout.

Normally, when a new process is created, it is to execute a program different from the one its parent is executing. A process can overlay itself with the memory image of another process, passing to the newly created process a set of parameters, using the system call *execve*. One parameter is the name of a file whose contents are in a format recognized by the system—either a binary-executable file, or a file that causes the execution of a specified interpreter program to process its contents.

A process may terminate by executing an *exit* system call, returning eight bits of exit status to its parent. If a process wants to communicate more than a single byte of information with its parent, it must either set up an interprocess-communication channel using pipes or sockets, or use an intermediate file. Interprocess communication is discussed more extensively in Chapter 10.

Figure 2.1 Process-management system calls.



A process can suspend execution until any of its child processes terminate using the *wait* system call, which returns its PID and exit status. A parent process can arrange to be notified by a signal when a child process exits or terminates abnormally. Using the *wait3* system call, the parent can retrieve information about the event that caused termination of the child process, and about resources consumed by the process during its lifetime. If a process is orphaned because its parent exits before it is finished, then the kernel arranges for its exit status to be passed back to a special system process (*init*; see Section 3.1 and Section 13.4).

The details of how the kernel creates and destroys processes are described in Chapter 5.

Processes are scheduled for execution according to a *process-priority* parameter. This priority is managed by a kernel-based scheduling algorithm. Users can influence the scheduling of a process by specifying a parameter (*nice*) that weights the overall scheduling priority, but in general are obligated to share the underlying CPU resources according to the kernel's scheduling policy.

Signals

The system defines a set of *signals* that may be delivered to a process. Signals in 4.3BSD are modeled after hardware interrupts. A process may specify a user-level subroutine to be a *handler* to which a signal should be delivered. When a signal is generated, it is blocked from further occurrence while it is being *caught* by the handler. Catching a signal involves saving the current process context and building a new one in which to run the handler. The signal is then delivered to the handler, which can either abort the process or return to the executing process (perhaps after setting a global variable). If the handler returns, the signal is unblocked and can be generated (and caught) again.

A process may instead specify that a signal is to be *ignored* or that a default action as determined by the kernel is to be taken. The default action of some signals is to terminate the process. This termination may be accompanied by creation of a *core file* that contains the current memory image of the process for use in postmortem debugging.

Some signals cannot be caught or ignored. These include SIGKILL, which is used to kill runaway processes, and the job-control signal SIGSTOP.

A process may choose to have signals delivered on a special stack so that sophisticated software stack manipulations are possible. For example, a language supporting coroutines needs to provide a stack for each coroutine. The language run-time system can allocate these stacks by dividing up the single stack provided by UNIX. If the kernel does not support a separate signal stack, the space allocated for each coroutine must be expanded by the amount of space required to catch a signal.

All signals have the same *priority*. If multiple signals are pending simultaneously, the order in which signals are delivered to a process is implementation-specific. Signal handlers execute with the signal that caused their invocation blocked, but other signals may yet occur. Mechanisms are provided that

processes may use to protect critical sections of code against the occurrence of specified signals.

The detailed design and implementation of signals is described in Section 4.7.

Process Groups

Processes are organized into *process groups*. Process groups are used to control access to terminals and to provide a means of distributing signals to collections of related processes. A process inherits its process group from its parent process. Mechanisms are provided by the kernel to allow a process to alter its process group or the process group of any of its descendents. Creating a new process group is easy; the value of a new process group is ordinarily that of the process identifier of the creating process.

The group of processes in a process group is sometimes referred to as a *job* and is manipulated by high-level system software, such as the shell. A common kind of job created by a shell is a *pipeline* of several processes connected by pipes such that the output of the first process is the input of the second, the output of the second is the input of the third, and so forth. The shell creates such a job by forking, setting the process-group identifier of the child process to the child's PID, then building up the rest of the pipeline by creating the rest of the processes in the job and setting their process-group identifiers to the same PID.

A user process can send a signal to each process in a process group as well as to a single process. A process in a specific process group may receive software interrupts affecting the group, causing the group to suspend or resume execution or to be interrupted or terminated.

A terminal has a process-group identifier assigned to it. This identifier is normally set to the identifier of a process group associated with the terminal. A job-control shell may create a number of process groups associated with the same terminal; the terminal is the *controlling terminal* for each process in these groups. A process may read from a descriptor for its controlling terminal only if the terminal's process-group identifier matches that of the process. If the identifiers do not match, the process will be blocked if it attempts to read from the terminal. By changing the process-group identifier of the terminal, a shell can arbitrate a terminal among several different jobs. This is called *job control* and is described along with process groups in Section 4.7.

2.5 Memory Management

Each process has its own private address space. The address space is divided into three logical segments: *text*, *data*, and *stack*. The text segment is read-only and contains the machine instructions of a program. The data and stack segments are both readable and writable. The data segment contains the initialized and uninitialized data portions of a program, whereas the stack segment holds the application's run-time stack. On most machines, the stack segment is automatically extended by the kernel as the process executes. A data segment may be

explicitly expanded or contracted by a system call being made, whereas the size of the text segment can be changed only when its contents are overlaid with data from the filesystem, or when debugging takes place. The initial contents of the segments of a child process are duplicates of the segments of a parent process.

The entire contents of a process address space need not be resident for a process to execute. If a process references a part of its address space that is not resident in main memory, the system *pages* the necessary information into memory. When system resources are scarce, the system uses a two-level approach to maintain available resources. If a modest amount of memory is available, the system will take memory resources away from processes if these resources have not been used recently. Should there be a severe resource shortage, the system resorts to *swapping* the entire context of a process to secondary storage. The *demand paging* and *swapping* performed by the system is effectively transparent to processes. A process may, however, advise the system about expected future memory utilization as a performance aid.

BSD Memory-Management Design Decisions

A requirement for 4.2BSD was that it support large sparse address spaces, mapped files, and shared memory. An interface was specified, called *mmap()*, that allowed unrelated processes to map a file into their address space. If multiple processes mapped the same file into their address space, changes to the address space by one process would be reflected in the area mapped by the other processes as well as in the file itself. Ultimately, 4.2BSD was shipped without the *mmap()* interface because of the pressures to make other features, such as networking, available.

Further development of the *mmap()* interface continued during the work on 4.3BSD. Over 40 companies and research groups participated in the discussions leading to the revised architecture that was described in the Berkeley Software Architecture Manual [Joy *et al.*, 1986]. Several of the companies have implemented the revised interface [Gingell *et al.*, 1987].

Once again, time pressures prevented 4.3BSD from providing an implementation of the interface. Although the latter could have been built into the existing 4.3BSD virtual-memory system, the developers decided not to do so because that implementation was nearly 10 years old. Furthermore, the original virtual-memory design was based on the assumption that computer memories were small and expensive, whereas disks were locally connected, fast, large, and inexpensive. Thus, the virtual-memory system was designed to be frugal with its use of memory at the expense of generating extra disk traffic. In addition, the 4.3BSD implementation was riddled with VAX memory-management hardware dependencies that impeded its portability to other computer architectures. Finally, the virtual-memory system was not designed to support the tightly coupled multiprocessors that are becoming increasingly common and important today.

Attempting to improve the old implementation incrementally seemed doomed to failure. A completely new design, on the other hand, could take advantage of large memories, conserve disk transfers, and have the potential to run on

multiprocessors. At the time of 4.3BSD's release, few of the implementations of virtual memory had been completed. Thus, the design alternatives were not yet tested and the optimal choices were still unknown. The hope is that the correct choices will become clear, and that a new virtual-memory system can be crafted for the next Berkeley UNIX system from the work of the new implementations.

Another issue with the virtual-memory system is the way that information is passed into the kernel when a system call is made. UNIX always copies data from the process address space into a buffer in the kernel. For read or write operations that are transferring large quantities of data, doing the copy can be time consuming. An alternative to doing the copying is to remap the process memory into the kernel. The 4.3BSD kernel always copies the data for several reasons:

- Often, the user data are not page aligned and are not a multiple of the hardware page length.
- If the page is taken away from the process, it will no longer be able to reference that page. Some programs depend on the data remaining in the buffer even after the latter have been written.
- If the process is allowed to keep a copy of the page (as with current 4.3BSD semantics), the page must be made *copy-on-write*. A copy-on-write page is one that is protected against being written by its being made read-only. If the process attempts to modify the page, the kernel gets a write fault. The kernel then makes a copy of the page that the process can modify. Unfortunately, the typical process will immediately try to write new data to the buffer, forcing the data to be copied anyway.
- The 4.3BSD virtual-memory design does not have provision for doing copy-on-write.
- When pages are remapped to new virtual-memory addresses, most memory-management hardware requires that the hardware address-translation cache be purged selectively. The cache purges are often slow. The net effect is that remapping is slower than copying for blocks of data less than 4 to 8 Kbyte.

The biggest incentive for memory mapping is the need for accessing big files and for passing large quantities of data between processes. The *mmap()* interface will provide a way for both of these tasks to be done without copying.

Memory Management Inside the Kernel

The 4.3BSD kernel has at least 10 different memory allocators. Some of them handle large blocks, some of them handle small, chained data structures, and others include information to describe I/O operations. Often, the allocations are for pieces of memory that are needed only for the duration of a single system call. In a user process, such short-term memory would be allocated on the run-time stack. Because the kernel has a limited run-time stack, it is not feasible to allocate even moderate-sized blocks of memory on it. Consequently, such memory must be

allocated through a more dynamic mechanism. For example, when the system must translate a pathname, it must allocate a 1-Kbyte buffer to hold the name.

Other blocks of memory must be more persistent than a single system call and thus could not be allocated on the stack even if there was space. An example is protocol-control blocks that remain throughout the duration of a network connection.

Demands for dynamic memory allocation in the kernel have increased as more services have been added. Each time a new type of memory allocation has been required, developers have written a specialized memory-allocation scheme to handle it. Often, the new memory-allocation scheme has been built on top of an older allocator. For example, the block-device subsystem provides a crude form of memory allocation through the allocation of empty buffers. The allocation is slow because of the implied semantics of finding the oldest buffer, pushing its contents to disk if they are dirty, and moving physical memory into or out of the buffer to create the requested size (see Section 7.5). To reduce the overhead, a “new” memory allocator is built for name translation that allocates a pool of empty buffers. It keeps them on a free list, so they can be quickly allocated and freed [McKusick *et al.*, 1985].

This memory-allocation method has three drawbacks. First, the new allocator can handle only a limited range of sizes. Second, it depletes the buffer pool as it steals memory intended to buffer disk blocks. Third, it creates yet another interface of which the kernel programmer must be aware.

A generalized memory allocator reduces the complexity of writing code inside the kernel. Rather than providing many semispecialized ways of allocating memory, the kernel provides a single, general-purpose allocator. With only a single interface, programmers need not concern themselves with trying to figure out the most appropriate way to allocate memory. A good general-purpose allocator helps to avoid the syndrome of inventing new special-purpose allocators.

The 4.3BSD Tahoe release introduced a memory allocator with an interface similar to the C library routines *malloc()* and *free()* that provide memory allocation to application programs [McKusick & Karels, 1988]. Like the C library interface, the allocation routine takes a parameter specifying the size of memory that is needed. The range of sizes for memory requests is not constrained; however, physical memory is allocated and is not paged. The free routine takes a pointer to the storage being freed, but does not require the size of the piece of memory being freed.

2.6 I/O System

The basic model of the UNIX I/O system is a sequence of bytes that can be accessed either randomly or sequentially. There are no *access methods* and no *control blocks* in a typical UNIX user process.

Different programs expect various levels of structure, but the kernel does not impose structure on I/O. For instance, the convention for text files is lines of

ASCII characters separated by a single newline character (the ASCII line-feed character), but the kernel knows nothing about this convention. For the purposes of most programs, the model is further simplified to being a stream of data bytes, or an *I/O stream*. It is this single common data form that makes the characteristic UNIX tool-based approach work [Kernighan & Pike, 1984]. An I/O stream from one program can be fed as input to almost any other program. (This kind of traditional UNIX I/O stream should not be confused with the Eighth Edition stream I/O system or with the System V, Release 3 STREAMS derived from them; both of those latter objects can be accessed in those systems as traditional I/O streams.)

Descriptors and I/O

UNIX processes use *descriptors* to reference I/O streams. A descriptor is a small unsigned integer. A *read* or *write* system call can be applied to a descriptor to transfer data. The *close* system call can be used to deallocate any descriptor.

Descriptors represent underlying objects supported by the kernel, and are created by system calls specific to the type of object. In 4.3BSD, three kinds of objects may be represented by descriptors: files, pipes, and sockets.

- A *file* is a linear array of bytes with at least one name. A file exists until all its names are explicitly deleted and no process holds a descriptor for it. A descriptor for a file is produced by opening that file's name with the *open* system call. I/O devices are accessed as files.
- A *pipe* is a linear array of bytes, as is a file, but it is used solely as an I/O stream, and it is unidirectional. It also has no name, and thus cannot be opened with *open*. Instead, it is created by the *pipe* system call, which returns two descriptors, one of which accepts input that is sent to the other descriptor reliably, without duplication, and in order.
- A *socket* is a transient object that is used for interprocess communication; it exists only as long as some process holds a descriptor referring to it. A socket is created by the *socket* system call, which returns a descriptor for it. There are different kinds of sockets that support various communication semantics, such as reliable delivery of data, preservation of message ordering, and preservation of message boundaries.

In UNIX systems prior to 4.2BSD, pipes are implemented using the filesystem; when sockets were introduced in 4.2BSD, pipes were reimplemented as sockets.

The kernel keeps a *descriptor table* for each process; this is a table that the kernel uses to translate the external representation of a descriptor—a small unsigned integer—into an internal representation. (The descriptor is merely an index into this table.) The descriptor table of a process is inherited from its parent, and thus access to the objects to which the descriptors refer also is inherited. The main ways a process can obtain a descriptor are by opening or creation of an object and by inheritance from the parent process. In addition, socket IPC allows passing descriptors in messages between unrelated processes.

line-feed
purposes
a bytes,
acteristic
am from
of tradi-
ream I/O
both of
ams.)

s a small
riptor to
or.
and are
kinds of

til all its
descrip-
tem call.

) stream,
ed with
descrip-
reliably,

ation; it
socket is
here are
, such as
ation of

esystem;
ckets.

that the
-a small
erely an
from its
is inher-
creation
cket IPC

Every valid descriptor-table entry in turn refers to a *file structure*, which contains a *file offset* in bytes from the beginning of the object. Read and write operations start at this offset, which is updated after each data transfer. For objects that permit random access, the file offset also may be set with the *lseek* system call. Ordinary files permit random access, and some devices do, as well. Pipes and sockets do not.

When a process terminates, the kernel reclaims all the descriptors that were in use by that process. If the process was holding the last reference to an object, the object's manager is notified so that it can do any necessary cleanup actions, such as final deletion of a file or deallocation of a socket.

Descriptor Management

Most processes expect three descriptors to be open already when they start running. These are descriptors 0, 1, 2, more commonly known as *standard input*, *standard output*, and *standard error*, respectively. Usually, all three are associated with the user's terminal by the login process (see Section 13.4) and are inherited through *fork* and *exec* by processes run by the user. Thus, a program can read what the user types by reading standard input, and the program can send output to the user's screen by writing to standard output. The standard error descriptor also is open for writing and is used for error output, whereas standard output is used for ordinary output.

These (and other) descriptors can be mapped to objects other than the terminal; this is called *I/O redirection*, and all the standard shells permit users to do it. The shell can direct the output of a program to a file by closing descriptor 1 (standard output) and opening the desired output file to produce a new descriptor 1. Standard input may similarly be redirected to come from a file by closing descriptor 0 and opening the file.

Pipes allow the output of one program to be input to another program without rewriting or even relinking of either program. Instead of descriptor 1 (standard output) of the source program being set up to write to the terminal, it is set up to be the input descriptor of a pipe. Similarly, descriptor 0 (standard input) of the sink program is set up to reference the output of the pipe instead of the terminal keyboard. The resulting set of two processes and the connecting pipe is known as a *pipeline*. Pipelines can be arbitrarily long series of processes connected by pipes.

The *open*, *pipe*, and *socket* system calls produce new descriptors with the lowest unused number usable for a descriptor. For pipelines to work, some mechanism must be provided to map such descriptors into 0 and 1. The *dup* system call creates a copy of a descriptor that points to the same file-table entry. The new descriptor is also the lowest unused one, but if the desired descriptor is closed first, *dup* can be used to do the desired mapping. Care must be taken, however, because if descriptor 1 is desired, and descriptor 0 happens also to have been closed, descriptor 0 will be the result. To avoid this problem, the *dup2* system call is provided; it is like *dup*, but it takes an additional argument specifying the number of the desired descriptor (if it was already open, *dup2* closes it before reusing it).

Files

A regular file in UNIX is a sequence of bytes with one or more names. The *open* system call takes as arguments the name of a file and a permission mode to specify whether the file should be open for reading, writing, or both. This system call also may be used to create a new, empty file (and there is also a system call *creat* for that purpose). The *read* and *write* system calls can be applied to the resulting descriptor, if the appropriate mode was set with *open*. A descriptor can be closed with the *close* system call.

The *link* system call takes the name of a file and makes another name for it. A filename may be removed with the *unlink* system call. When the last name for a file is removed (and the last process that has the file open closes it), the file is deleted.

Files are organized hierarchically in *directories*. A directory is a type of file, but, in contrast to regular files, a directory has a structure imposed on it by the system. A process can read a directory as it would an ordinary file, but only the system is permitted to modify a directory. Directories are created by the *mkdir* system call and are removed by the *rmdir* system call. Before 4.2BSD, the *mkdir* and *rmdir* system calls were implemented by a series of *link* and *unlink* system calls being done. There were three reasons for adding systems calls explicitly to create and delete directories:

- The operation could be made atomic. If the system crashed, the directory would not be left half constructed, as could happen when using a series of link operations.
- When a networked filesystem is being run, the creation and deletion of files and directories need to be specified atomically so that they can be serialized.
- Non-UNIX filesystems may not support link operations. Although they might support the concept of directories, they probably would not create and delete the directories with links, as the UNIX filesystem does. Consequently, they could create and delete directories only if explicit directory create and delete requests were presented.

The *chown* system call sets the owner and group of a file, and *chmod* changes protection attributes. *Stat* applied to a filename can be used to read back such properties of a file. The *fchown*, *fchmod*, and *fstat* system calls are applied to a descriptor instead of to a filename to do the same set of operations. The *rename* system call may be used to give a file a new name in the filesystem, replacing one of the file's old names. Like the directory-creation and directory-deletion operations, the *rename* system call was added to 4.2BSD to provide atomicity to name changes in the local filesystem. Later, it proved useful explicitly to export renaming operations to foreign filesystems and over the network.

The *truncate* system call was added to 4.2BSD to allow files to be shortened to an arbitrary offset. The call was added primarily in support of the Fortran run-time library which has the semantics such that the end of a random-access file is

set to be wherever the program last accessed that file. Without the *truncate* system call, the only way to shorten a file was to copy the part that was desired to a new file, delete the old file, then rename the copy to the original name. Besides being slow, the library could potentially fail on a full filesystem.

Once the filesystem had the ability to shorten files, the kernel took advantage of that ability to shorten large empty directories. The advantage of shortening empty directories is that it reduces the time spent in the kernel searching them when names are being created or deleted.

Some other UNIX systems provide *FIFOs*. A FIFO is essentially a pipe that has a name in the filesystem and that does not vanish when the last descriptor referring to it is closed. Because POSIX has FIFOs, future 4BSD releases may also include them. See Section 1.2 for filesystem facilities adopted by POSIX from 4.3BSD.

Devices

Hardware devices have filenames, and may be accessed by the user by the same system calls as are used for regular files. The kernel can distinguish a *device special file* or *special file* and determine to what device it refers, but most processes do not need to do this. Terminals, printers, and tape drives are all accessed as though they were streams of bytes, like UNIX disk files. Thus, device dependencies and peculiarities are kept in the kernel as much as possible, and even in the kernel most of them are segregated in the device drivers.

Device special files are created by the *mknod* system call. There is an additional system call, *ioctl*, for manipulating the underlying device parameters of special files. The operations that can be done differ for each device. This system call allows the special characteristics of devices to be accessed, rather than overloading the semantics of other system calls. For example, there is an *ioctl* on a tape drive to write an end-of-tape mark instead of there being a special or modified version of *write*.

Socket IPC

The 4.2BSD kernel introduced an IPC mechanism more flexible than pipes, based on *sockets*. A socket is an endpoint of communication referred to by a descriptor, just like a file or a pipe. Two processes can each create a socket, and then connect those two endpoints to produce a reliable byte stream. Once connected, the descriptors for the sockets can be read or written by processes, just as the latter would do with a pipe. The transparency of sockets allows the kernel to redirect the output of one process to the input of another process residing on another machine. A major difference between pipes and sockets is that pipes require a **common parent** process to set up the communications channel. A connection **between sockets** can be set up by two unrelated processes, possibly residing on **different machines**.

The **socket mechanism** requires extensions to the traditional UNIX I/O system calls to provide the associated naming and connection semantics. Rather than

overloading the existing interface, the developers used the existing interfaces to the extent that the latter worked without being changed, and designed new interfaces to handle the added semantics. The *read* and *write* system calls were used for byte-stream type connections, but six new system calls were added to allow sending and receiving addressed messages such as network datagrams. The system calls for writing messages include *send*, *sendto*, and *sendmsg*. The system calls for reading messages include *recv*, *recvfrom*, and *recvmsg*. In retrospect, the first two in each class are special cases of the others; *recvfrom* and *sendto* probably should have been added as library interfaces to *recvmsg* and *sendmsg* respectively.

Scatter/Gather I/O

In addition to the traditional *read* and *write* system calls, 4.2BSD introduced the ability to do scatter/gather I/O. Scatter input uses the *readv* system call to allow a single read to be placed in several different buffers. Conversely, the *writew* system call allows several different buffers to be written in a single atomic write. Instead of passing a single buffer and length parameter, as is done with *read* and *write*, the process passes in a pointer to an array of buffers and lengths, along with a count describing the size of the array.

This facility allows buffers in different parts of a process address space to be written atomically without the need to copy them to a single contiguous buffer. Atomic writes are necessary in the case where the underlying abstraction is record-based, such as tape drives that output a tape block on each write request. It is also convenient to be able to read a single request into several different buffers (such as a record header into one place and the data into another). Although the ability to scatter data can be entirely simulated by reading the data into a large buffer and then copying the pieces to their intended destinations, the cost of memory-to-memory copying in such cases often would more than double the running time of the affected program.

Just as *send* and *recv* could have been implemented as library interfaces to *sendto* and *recvfrom*, it would have also been possible to simulate *read* with *readv* and *write* with *writew*. However, *read* and *write* are used so much more frequently that the added cost of simulating them would not have been worthwhile.

2.7 Filesystem

A regular file is a linear array of bytes, and can be read and written starting at any byte in the file. The kernel distinguishes no record boundaries in regular files, although many programs recognize line-feed characters as distinguishing the ends of lines, and other programs may impose other structure. No system-related information about a file is kept in the file itself, but the filesystem stores a small amount of ownership, protection, and usage information with each file.

A *filename* component is a string of up to 255 characters. These filenames are stored in a type of file called a *directory*. The information in a directory about

aces to
/ inter-
re used
allow
he sys-
system
ect, the
prob-
respec-

ced the
allow a
ev sys-
write.
ad and
ng with

e to be
buffer.
tion is
uest. It
buffers
gh the
a large
cost of
he run-

aces to
a *readv*
ore fre-
hile.

g at any
ar files.
he ends
d infor-
a small

enames
y about

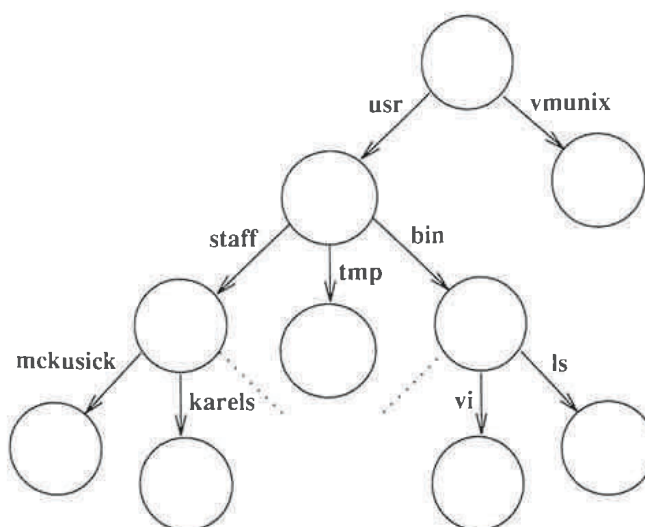
a file is called a *directory entry* and includes, in addition to the filename, a pointer to the file itself. Directory entries may refer to other directories as well as to plain files. A hierarchy of directories and files is thus formed, and is called a *filesystem*; a small one is shown in Fig. 2.2. Directories may contain subdirectories, and there is no inherent limitation to the depth with which directory nesting may occur. To protect the consistency of the filesystem, the kernel does not permit processes to write directly into directories. A filesystem may include not only plain files and directories, but also references to other objects, such as devices and sockets.

The filesystem forms a tree, the beginning of which is the *root directory*, sometimes referred to by the name **slash**, spelled with a single solidus character (/). The root directory contains files—in our example in Fig. 2.2, **vmunix**, a copy of the kernel-executable object file. It also contains directories—in this example, the **usr** directory. Within the **usr** directory is the **bin** directory, which mostly contains executable object code of programs, such as the files **ls** and **vi**.

A process identifies a file by specifying that file's *pathname*, a string composed of zero or more *filenames* separated by slash (/) characters. The kernel associates two directories with each process for use in interpreting pathnames. A process's *root directory* is the topmost point in the filesystem that the process can access; it is ordinarily set to the root directory of the entire filesystem. A pathname beginning with a slash is called an *absolute pathname* and is interpreted by the kernel starting with the process's root directory.

A pathname that does not begin with a slash is called a *relative pathname* and is interpreted relative to the *current working directory* of the process. (This directory is also known by the shorter names *current directory* or *working directory*.)

Figure 2.2 A small filesystem tree.



The current directory itself may be referred to directly by the name *dot*, spelled with a single period (.). The filename *dot-dot* (..) refers to a directory's parent directory. The root directory is its own parent.

A process may set its root directory with the *chroot* system call, and its current directory with the *chdir* system call. Any process may do *chdir* at any time, but *chroot* is permitted only to the superuser, and is normally used in setting up restricted access to the system.

Using the filesystem shown in Fig. 2.2, if a process has the root of the filesystem as its root directory, and */usr* as its current directory, it can refer to the file *vi* either from the root with the absolute pathname */usr/bin/vi*, or from its current directory with the relative pathname *bin/vi*.

System utilities and databases are kept in certain well-known directories. Part of the well-defined hierarchy includes a directory that contains the *home directory* for each user; for example, */usr/staff/mckusick* and */usr/staff/karels* in Fig. 2.2. When users log in, the current working directory of their shell is set to the home directory. Within their home directories, users can create directories as easily as they can regular files. Thus, a user can build arbitrarily complex subhierarchies.

The user usually knows of only one filesystem, but the system may know that this one virtual filesystem is really composed of several physical filesystems, each on a different device. A physical filesystem may not span multiple hardware devices. Since most physical disk devices are divided into several logical devices, there may be more than one filesystem per physical device, but no more than one per logical device. One filesystem—the filesystem that anchors all absolute pathnames—is called the *root filesystem*, and is always available. Others may be mounted; that is, they may be integrated into the directory hierarchy of the root filesystem. References to a directory that has a filesystem mounted on it are transparently converted by the kernel into references to the root directory of the mounted filesystem.

Newly created files are assigned the user identifier of the process that created them and the group identifier of the directory in which they were created. A three-level access-control mechanism is provided for the protection of files. These three levels specify the accessibility of a file to

- The user who owns the file
- The group that owns the file
- Everyone else

Each level of access has separate indicators for read permission, write permission, and execute permission.

Files are created with zero length and may grow when they are written. While a file is open, the system maintains a pointer into the file indicating the current location in the file associated with the descriptor. This pointer can be moved about in the file in a random-access fashion. Files may have *holes* in them.

spelled
s parent

and its
r at any
i setting

filesys-
e file vi
current

ctories.
e *home*
arels in
is set to
ories as
omplex

ow that
is, each
rdware
levices,
an one
bsolute
may be
he root
e trans-
of the

created
ted. A
f files.

ission.

written.
ng the
can be
1 them.

Holes are void areas in the linear extent of the file where data have never been written. A process can create these holes by positioning the pointer past the current end-of-file and writing. When read, holes are treated by the system as zero-valued bytes.

Earlier UNIX systems had a limit of 14 characters per filename component. This limitation was often a problem. For example, besides the natural desire of users to give files long descriptive names, a common way of forming filenames is as **basename.extension**, where the extension (indicating the kind of file, such as **.c** for C source or **.o** for intermediate binary object) is one to three characters, leaving 10 to 12 characters for the basename. Source-code-control systems and editors usually take up another two characters, either as a prefix or a suffix, for their purposes, leaving eight to 10 characters. It is easy to use 10 or 12 characters in a single English word as a basename (e.g., "multiplexer").

It is possible to keep within these limits, but it is inconvenient or even dangerous, because other UNIX systems accept strings longer than the limit when creating files, but then *truncate* to the limit. A C language source file named **multiplexer.c** (already 13 characters) might have a source-code-control file with **s.** prepended, producing a filename **s.multiplexer** that is indistinguishable from the source-code-control file for **multiplexer.ms**, a file containing **troff** source for documentation for the C program. The contents of the two original files could easily get confused with no warning from the source-code-control system. Careful coding can detect this problem, but the long filenames first introduced in 4.2BSD practically eliminate it.

2.8 Devices

Hardware devices can be categorized as either *structured* or *unstructured*; these are known as *block* or *character* devices, respectively. Processes typically access devices through *special files* in the filesystem. I/O operations to these files are handled by kernel-resident software modules termed *device drivers*. Most network-communication hardware devices are accessible only through the interprocess-communication facilities and do not have special files in the filesystem name space. This is because the *raw-socket* interface provides a more natural interface than does a special file.

Structured or block devices are typified by disks and magnetic tapes, and include most random-access devices. The kernel supports read-modify-write-type buffering actions on block-oriented structured devices to allow the latter to be read and written in a totally random byte-addressed fashion, like regular files. Filesystems are created on block devices.

Unstructured devices are those devices that do not support a block structure. Familiar unstructured devices are communication lines, raster plotters, and unbuffered magnetic tapes and disks. Unstructured devices typically support large block I/O transfers.

Unstructured files are called *character devices* because the first of these to be implemented were terminal device drivers. The kernel interface to the driver for these devices proved convenient for other devices that were not block structured.

2.9 Terminals

Terminals support the standard system I/O operations as well as a collection of terminal-specific operations to control input-character editing and output delays. At the lowest level are the terminal device drivers that control the hardware terminal ports. Terminal input is handled according to the underlying communication characteristics, such as baud rate, and a set of software-controllable parameters, such as parity checking.

Layered above the terminal device drivers are line disciplines that provide various degrees of character processing. The default line discipline usually is used when a port is being used for an interactive login. The default line discipline has three modes: *raw*, *cbreak*, or *cooked*. In raw mode, input is passed through to the reading process immediately and without interpretation. In cbreak mode, the terminal handler interprets only input characters that cause the generation of signals or output flow control; all other characters are passed through to the process as in raw mode. In cooked mode, input is processed to provide standard line-oriented editing functions, and input is presented to a process on a line-by-line basis.

On output, the terminal handler provides some simple formatting services. These services include

- Converting the line-feed character to the two-character return–line-feed sequence
- Inserting delays after certain standard control characters
- Expanding tabs
- Providing translation for uppercase-only terminals
- Displaying echoed nongraphic ASCII characters as a two-character sequence of the form “^X” (i.e., the ASCII caret character followed by a printable ASCII character related to the control character)

Each of these formatting services can be disabled individually by a process through control requests.

2.10 Interprocess Communication

Interprocess communication in 4.3BSD is organized in *communication domains*. Domains currently supported include the *UNIX domain*, for communication

between processes executing on the same machine; the *internet domain*, for communication between processes using the TCP/IP protocol suite (perhaps within the DARPA Internet or NSFnet); and the *XNS domain*, for communication between processes using the XEROX Network Systems (XNS) protocols.

Within a domain, communication takes place between communication endpoints known as *sockets*. As mentioned in Section 2.6, the *socket* system call creates a socket and returns a descriptor; other IPC system calls are described in Chapter 10. Each socket has a type that defines its communications semantics; these include properties such as reliability, ordering, and prevention of duplication of messages.

Each socket has a *communication protocol* associated with it. This protocol provides the semantics required by the socket type. Applications may request a specific protocol when creating a socket, or may allow the system to select a protocol that is appropriate for the type of socket being created.

Sockets may have addresses bound to them. The form and meaning of socket addresses are dependent on the communication domain in which the socket is created. Binding a name to a socket in the UNIX domain causes a file to be created in the filesystem.

Normal data transmitted and received through sockets are untyped. Data-representation issues are the responsibility of libraries built on top of the interprocess-communication facilities. In addition to normal data, communication domains may support the transmission and reception of specially typed data, termed *access rights*. The UNIX domain, for example, uses this facility to pass descriptors between processes.

Networking implementations on UNIX before 4.2BSD usually worked by overloading the character device interfaces. One goal of the socket interface was that it should be possible for naive programs to work without change on stream-style connections. Such programs can work only if the *read* and *write* systems calls are unchanged. Consequently, the original interfaces were left intact and were made to work on stream-type sockets. A new interface was added for more complicated sockets, such as those used to send datagrams, with which a destination address must be presented with each *send* call.

Another benefit of the new interface is that it is highly portable. Shortly after a test release was available from Berkeley, the socket interface had been ported to System III by a UNIX vendor (although AT&T never adopted the socket interface or implementation, deciding instead to use the Eighth Edition stream mechanism). The socket interface was also ported to run in many Ethernet boards by vendors such as Excelan and Interlan that were selling into the PC market, where the machines were too small to run networking in the main processor.

2.11 Network Communication

Some of the communication domains supported by the *socket* IPC mechanism provide access to network protocols. These protocols are implemented as a

separate software layer logically below the socket software in the kernel. The kernel provides many ancillary services, such as buffer management, message routing, standardized interfaces to the protocols, and interfaces to the network interface drivers for the use of the various network protocols. These services and interfaces are described in Chapter 11.

At the time that 4.2BSD was being implemented, there were many networking protocols in use or under development, each with its own strengths and weaknesses. There was no clearly superior protocol or protocol suite. By supporting multiple protocols, 4.2BSD could provide interoperability and resource sharing among the diverse set of machines that was available in the Berkeley environment. Multiple protocol support also provides for future changes. Today's protocols designed for 10 Mbit/s Ethernet are likely to be inadequate for tomorrow's 100 Mbit/s fiber-optic nets. Consequently, the network-communication layer is designed to support multiple protocols. New protocols are added to the kernel without the support for older protocols being affected. Older applications can continue to operate using the old protocol over the same physical network as is used by newer applications running with a newer network protocol.

2.12 Network Implementation

The first protocol suite implemented in 4.2BSD was DARPA's Transmission Control Protocol/Internet Protocol (TCP/IP). CSRG chose TCP/IP as the first network to incorporate into the socket IPC framework because a 4.1BSD-based implementation was publicly available from a DARPA-sponsored project at Bolt, Beranek, and Newman (BBN). That was an influential choice: The 4.2BSD implementation is the main reason for the extremely widespread use of this protocol suite. Subsequent performance and capability improvements to the TCP/IP implementation have also been widely adopted. The TCP/IP implementation is described in detail in Chapter 12.

The release of 4.3BSD added the Xerox Network Systems (XNS) protocol suite, partly building on work done at the University of Maryland and Cornell University. This suite was needed to connect isolated machines that could not communicate using TCP/IP.

The ISO protocol suite will be the next network implementation added to future versions of 4.3BSD because of its increasing visibility both within and outside the United States.

2.13 System Operation

Bootstrapping mechanisms are used to start the system running. First, the UNIX kernel must be loaded into the main memory of the processor. Once loaded, it must go through an initialization phase to set the hardware into a known state.

The
message
network
s and

orking
s and
y sup-
source
keley
anges.
ate for
work-
ocols
ected.
same
network

1 Con-
network
emen-
ranek,
station
Subse-
ntation
detail

otocol
Cornell
ld not

ded to
id out-

UNIX
ded, it
1 state.

Next, the kernel must do autoconfiguration, a process that finds and configures the peripherals that are attached to the processor. The system begins running single-user while a start-up script does disk checks and starts the accounting and quota checking. Finally, the start-up script starts the general system services and brings up the system to full multiuser operation.

During multiuser operation, processes wait for login requests on the terminal lines and network ports that have been configured for user access. When a login request is detected, a login process is spawned and user validation is done. When the login validation is successful, a login shell is created from which the user can run additional processes.

Exercises

- 2.1 How does a user process request a service from the kernel?
- 2.2 How are data transferred between a process and the kernel? What alternatives are available?
- 2.3 How does a process access an I/O stream? List three types of I/O streams.
- 2.4 What are the four steps in the lifecycle of a process?
- 2.5 Why are process groups provided in 4.3BSD?
- 2.6 What are the machine-dependent functions of the kernel?
- 2.7 Describe the difference between an absolute and a relative pathname.
- 2.8 List the three system calls that were required to create a new directory **foo** in the current directory before the addition of the *mkdir* system call. Give three reasons why the *mkdir* system call was added to 4.2BSD.
- 2.9 Define scatter-gather I/O. Why is it useful?
- 2.10 What is the difference between a block and a character device?
- 2.11 List five functions provided by a terminal driver.
- 2.12 What is the difference between a pipe and a socket?
- 2.13 Describe how to create a group of processes in a pipeline.
- *2.14 Explain the difference between interprocess communication and networking.

References

- Accetta *et al.*, 1986.
M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, & M. Young,
"Mach: A New Kernel Foundation for UNIX Development," Technical
Report, Carnegie Mellon University, Pittsburgh, PA (August 1986).

Cheriton, 1988.

D. R. Cheriton, "The V Distributed System," *Comm ACM* **31**(3), pp. 314–333 (March 1988).

Ewens *et al.*, 1985.

P. Ewens, D. R. Blythe, M. Funkenhauser, & R. C. Holt, "Tunis: A Distributed Multiprocessor Operating System," *USENIX Association Conference Proceedings*, pp. 247–254 (June 1985).

Fitzgerald & Rashid, 1986.

R. Fitzgerald & R. F. Rashid, "The Integration of Virtual Memory Management and Interprocess Communication in Accent," *ACM Transactions on Computer Systems* **4**(2), pp. 147–177 (May 1986).

Gingell *et al.*, 1987.

R. A. Gingell, J. P. Moran, & W. A. Shannon, "Virtual Memory Architecture in SunOS," *USENIX Association Conference Proceedings*, pp. 81–94 (June 1987).

Joy *et al.*, 1986.

W. N. Joy, R. S. Fabry, S. J. Leffler, M. K. McKusick, & M. J. Karels, "Berkeley Software Architecture Manual, 4.3BSD Edition," pp. 6:1–43 in *UNIX Programmer's Supplementary Documents, Volume 1, 4.3 Berkeley Software Distribution, Virtual VAX-11 Version*, USENIX Association, Berkeley, CA (1986).

Kernighan & Pike, 1984.

B. W. Kernighan & R. Pike, *The UNIX Programming Environment*, Prentice-Hall, Englewood Cliffs, NJ (1984).

McKusick *et al.*, 1985.

M. K. McKusick, M. Karels, & S. Leffler, "Performance Improvements and Functional Enhancements in 4.3BSD," *USENIX Association Conference Proceedings*, pp. 519–531 (June 1985).

McKusick & Karels, 1988.

M. K. McKusick & M. J. Karels, "Design of a General Purpose Memory Allocator for the 4.3BSD UNIX Kernel," *USENIX Association Conference Proceedings*, pp. 295–304 (June 1988).

Ritchie, 1988.

D. M. Ritchie, "Early Kernel Design," private communication (March 1988).

CHAPTER 3

Kernel Services

3.1 Kernel Organization

The UNIX kernel provides services to user processes. Processes access most services directly through system calls. Other services, such as process scheduling and memory management, are implemented as UNIX processes that execute in kernel mode or as routines that execute periodically within the kernel. In this chapter, we shall describe how kernel services are provided to user processes, and explain the ancillary processing that is performed by the kernel in normal operation. Following this description, we shall introduce some of the basic kernel services and shall describe their implementation.

System Activities

Each entry into the kernel is termed a *system activity*. These activities can be categorized according to the event or action that initiates them:

- System calls
- Hardware interrupts
- Hardware traps
- Software-initiated traps or interrupts

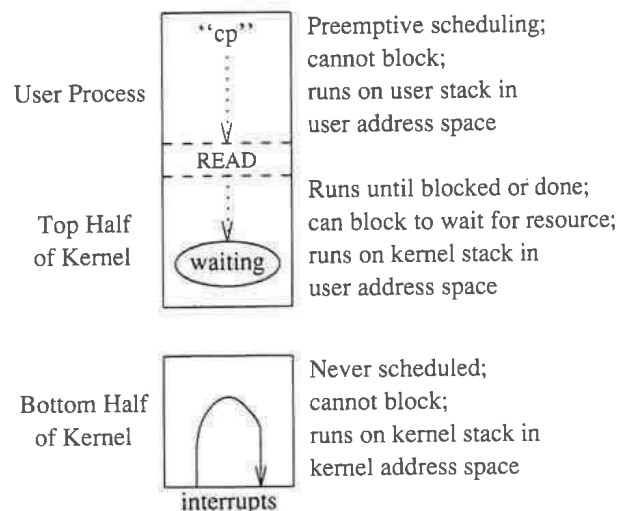
System calls usually are a special case of the last category—the machine instruction used to initiate a system call typically causes a hardware trap that is treated specially by the kernel. Hardware interrupts arise from external events, such as an I/O device needing attention. The kernel depends on the presence of a real-time clock or interval timer to maintain the current time of day, to drive process scheduling, and to initiate the execution of system timeout functions. Hardware

interrupts occur *asynchronously* and rarely relate to the context of the process that is currently *executing*. Hardware traps, on the other hand, usually are unexpected, but *are* related to the current executing process. Examples of hardware traps are those generated as a result of incorrect *arithmetic* calculations, such as dividing by zero. Finally, software-initiated traps *or* interrupts are used by the system to force the scheduling of a system activity at an opportune time. An example of this type of event is the *asynchronous system trap* (AST) that is posted by the system while it is executing in kernel mode, but that is not delivered until user mode is reentered. (ASTs are used to force process rescheduling.)

Run-Time Organization

The run-time structure of the kernel is divided into a *top half* and a *bottom half*, as shown in Fig. 3.1. The top half of the kernel provides services to applications in response to system calls or traps. This software can be thought of as a library of routines that are shared among all UNIX processes. The top half of the kernel executes in a privileged execution mode in which it has access both to kernel data structures and to the context of the user-level process that it supports. Each process has an area of memory reserved for process-specific information used by the kernel when the process accesses system services. This information includes a small stack for use by the kernel while executing system calls for the process. Below this stack is the *user area* that contains long-term information about the state of the process. This information includes the unique identifiers associated with the process. It also records information about the maximum and current resource utilization and about the rights and privileges associated with the

Figure 3.1 Run-time structure of the kernel.



ess that
pected,
aps are
ding by
o force
is type
n while
is reen-

half, as
tions in
rary of
nel exe-
nel data
ach pro-
d by the
cludes a
process.
bout the
sociated
current
with the

process. Pending and future external events are recorded in the user area, along with the action to take when they are ready to be delivered to the process. Finally, the user area records all the descriptors held by the process.

The bottom half of the kernel is a collection of routines that are invoked to handle hardware interrupts and traps that are unrelated to the current process. Interrupts may be delivered by I/O devices or by other hardware devices, such as a real-time clock. The kernel assumes that interrupts are prioritized at least to the extent that hardware facilities are available to block the delivery of interrupts. Whereas the VAX assumes there are distinct hardware priority levels for different kinds of interrupts, 4BSD also runs on architectures such as the Perkin Elmer, where interrupts are all at the same priority, or the ELXSI, where there are no interrupts in the traditional sense.

Activities in the bottom half of the kernel occur *asynchronously*, and the software cannot depend on having a specific process running at the time of an interrupt. In particular, some per-process state may not be available, and the run-time stack used by the top half of the kernel should not be altered by routines in the bottom half. Instead, the bottom half of the kernel typically runs on a system-wide *interrupt stack*.

The kernel is never preempted to run another process while executing the code to do a system call. Its execution may be interrupted, however, by activities done by the bottom half of the kernel. It must explicitly give up the processor to wait for a resource or run to completion. The top half can block entry to routines in the bottom half of the kernel by setting the *processor priority level* to a value that blocks out the appropriate interrupts. This mechanism is used, for example, to ensure the consistency of data structures shared between the top and bottom halves.

Processes cooperate in the sharing of system resources, such as the CPU. The top half of the kernel can voluntarily relinquish control of the processor when it must wait for an event or for access to a shared resource. The top and bottom halves of the kernel also work together in implementing certain system operations, such as I/O. Typically, the top half will start an I/O operation, relinquish the processor, and then wait for notification from the bottom half when the I/O request has completed.

System Processes

All UNIX processes originate from a single process that is handcrafted by the kernel at startup. Three processes are always expected to exist. Two of these processes, termed *kernel processes*, function wholly within the kernel. Kernel processes execute code that is compiled into the kernel's load image and operate with the kernel's privileged execution mode. The third process is the first process to execute a program in user mode; it serves as the parent process to all subsequent processes.

The two kernel processes are termed the *swapper* and the *pagedaemon*. The *swapper*, historically process 0, is responsible for scheduling the transfer of whole processes between main memory and secondary storage when system resources

are low. The *pagedaemon*, process 2, is specific to Berkeley versions of UNIX and is responsible for writing parts of the address space of a process to secondary storage in support of the paging facilities of the virtual-memory system. The third process created by the system is the *init* process (process 1). This process performs administrative tasks such as spawning login processes for each terminal on a machine and handling the orderly shutdown of a system from multiuser to single-user operation. The actions of the *init* process are embodied in a program that is independent of the kernel (see Section 13.4).

Entry to the Kernel

When a process enters the kernel through a system call, a trap, or an interrupt, the kernel must first ensure that it saves all the machine state before it begins to service the event. For the VAX, the machine state that must be saved includes the 16 general-purpose registers and the processor status longword. If the machine state were not fully saved, the kernel would change values in the currently executing program in unexpected and improper ways. Since interrupts may occur between any two user-level instructions, and because they may be completely unrelated to the currently executing process, incompletely saved state would cause otherwise correct programs to fail in mysterious and unreproducible ways even while not doing system calls.

The exact sequence of events required to save the process state is highly machine-dependent, although the VAX provides a good example of the general procedure. When a trap or system call occurs, the following events occur:

- The hardware switches into kernel mode, so that memory-access checks are made with kernel privilege, references to the stack pointer use the kernel's stack pointer, and privileged instructions can be executed.
- The program counter and processor status longword are pushed onto the per-process kernel stack.
- A code showing the type of trap (or system call number) is pushed onto the stack.
- An assembly-language routine saves the general-purpose registers.

This preliminary state saving allows the kernel then to call a C routine that can freely use the machine registers as any other C routine would, without concern about destroying an unsuspecting process's state. There are several major kinds of handlers, corresponding to particular kinds of kernel entries:

- *Syscall()* for a system call
- *Trap()* for a trap
- An appropriate device-driver interrupt handler for a device interrupt

The parameters to the handler include the user's stack pointer, the program counter, and the code showing the trap type.

Return from the Kernel

On completion of its task, the handler returns. An assembly-language routine then restores the register state and executes a *return from interrupt* instruction. The *return from interrupt* instruction reverses the process of entering the kernel by

- Restoring the program counter and program status longword previously pushed onto the stack
- Changing to user mode, so that future references to the stack pointer use the user's stack pointer, privileged instructions may not be executed, and memory-access checks are done with user-level privilege

Execution resumes at the next instruction in the user's process.

3.2 System Calls

The most frequent trap into the kernel (after clock processing) is a request to do a system call. Good performance requires the kernel to minimize the overhead in fielding and dispatching a system call. The system-call dispatcher must do the following work to initiate a system call:

- Look up the number of parameters in the system call
- Verify that the parameters are located at a valid user address, then copy them from the user process into the kernel
- Prepare for the possibility of an interrupted system call
- Call a specific kernel routine to do the system call

Result Handling

Eventually, a system call returns to the calling process, either successfully or unsuccessfully. On the VAX, success or failure is indicated by the carry bit in the user process's program status longword: If it is zero, the return was successful; else, it was unsuccessful. On the VAX and many other machines, return values of C functions are passed back through general-purpose register 0. On a successful system call, the kernel system-call handler replaces the saved value of that register with a return value. On an unsuccessful call, the system-call handler takes an error code from the kernel variable `u.u_error` (where it has been left by deeper kernel routines for that system call) and puts that code in the register—this is a place where the kernel deliberately modifies the state of a process. The system call C library routine moves that value to the global variable `errno` and sets the

register to -1. The calling process is supposed to notice the value of the register and then examine *errno* on an error. This mechanism involving the carry bit and the global variable *errno* exists for historical reasons derived from the PDP-11.

There are two kinds of unsuccessful returns from a system call: those where kernel system-call-handling routines discover an error, and those where a system call is interrupted. A system call can be interrupted when it has relinquished the processor to wait for an event that may not occur for a long time (such as waiting for terminal input), and a signal arrives in the interim. Before the signal can be posted, the system call must be aborted by doing a nonlocal goto back to the dispatcher. When signal handlers are set up by a process, they must specify whether system calls that they interrupt should be restarted, or aborted with an *interrupted system call* error.

The dispatcher determines what to do when the nonlocal goto occurs, by examining a per-process global variable. If the process has requested that the system call be aborted, the dispatcher arranges to return an error, as described previously. If the system call is to be restarted, however, the dispatcher calculates how to make the process's program counter point to the machine instruction that caused the system-call trap into the kernel. This calculation is necessary because the program-counter value that was saved when the system-call trap was done is for the beginning of the system-call result-handling code. The dispatcher replaces the saved program-counter value with the newly calculated value. The next time the process is scheduled, it will be forced to handle the signal. When it returns from its signal handler, it will then resume at the program-counter value that the dispatcher provided, so the process will reexecute the same system call.

Restarting a system call by backing up the program counter has certain implications. Most important is that the kernel must ensure that it does not modify any of the input parameters. Second, it must ensure that the system call has not partially completed. For example, if a partial line has been read from the terminal, the read must return with a short count. If the call were restarted, the partially read line would be inappropriately overwritten.

Returning from a System Call

While the system call is running, a signal may be posted to the process, or another process may attain a higher scheduling priority. After the system call completes, the dispatcher checks to see whether either of these events has occurred.

The dispatcher first checks for a posted signal. These include signals that caused the system call to abort, as well as signals that arrived while a system call was in progress, but were held pending while the system call completed. Signals that are ignored either by default or by an explicit setting are never posted to the process. Signals that have a default action have that action taken before the process is able to run again (i.e., the process is killed or stopped as appropriate). If a signal is to be caught (and is not currently blocked), the dispatcher arranges to have the appropriate signal handler called rather than have the process return directly from the system call. Only after the handler returns normally will the process resume execution at system-call return (or system-call execution, if the system call is to be restarted).

After checking for posted signals, the dispatcher checks to see whether another process has a higher priority than does the currently running one. If such a process exists, the dispatcher calls the context-switch routine to arrange to have the higher-priority process run. At a later time, the current process will attain the highest priority, and will resume execution by returning from the system call to the user process.

If a process has requested the system to do profiling, the dispatcher calculates the amount of time that has been spent in the system call. The time is calculated by taking the difference in system time accounted to the process between its entry and exit from the dispatcher. The system time is charged to the routine in the user's process that did the system call.

3.3 Traps and Interrupts

Traps, like system calls, occur synchronously for a process. Traps can occur because of unintentional errors, such as division by zero, or because of indirection through an invalid pointer; the process is made aware of the problem either by catching a signal or by being terminated. Traps can also occur because of a page fault, in which case the system arranges to make the page available and then restarts the process without the process ever being aware that the fault occurred.

The trap handler is invoked like the system-call dispatcher. First, the process state is saved. Next, the trap handler determines the trap type, then arranges to post a signal or to cause a page-in as appropriate. Finally, it checks for pending signals and exits identically to the system-call handler.

I/O Device Interrupts

Interrupts from I/O devices are handled by *device-driver* modules that are included in the kernel at the time a system is built. Interrupts from other devices are handled by interrupt routines that are a standard part of the kernel. These routines handle a console terminal interface, a real-time clock, and several software-initiated interrupts used by the system for low-priority clock processing and for the networking facilities.

Unlike traps and system calls that are executed synchronously by a process, device interrupts occur asynchronously. Usually, the process that started the operation that eventually caused the interrupt is not even running. As with traps and system calls, the entire machine state must be saved, since any inadvertent changes would cause irreproducible errors in the currently running process. Notification to the process that requested the service must be done by requesting that it be run; it will then notice that the operation that it started has finished.

Device-interrupt handlers run only on demand; they are never **scheduled** by the kernel. Unlike system calls, interrupt handlers do not have a per-process context. Interrupt handlers cannot depend on using any of the context of the currently running process (i.e., the process's **user structure**). The stack normally used by the kernel is part of a process context. On some systems, interrupts are caught on the

unused part of the kernel stack of whichever process is currently running. This approach requires that all the per-process kernel stacks be large enough to handle both a system call and the deepest nesting of interrupts. On the VAX, there is a single interrupt stack used by all device interrupts. When an interrupt occurs, the hardware switches to the interrupt stack before beginning to handle the interrupt. Having a separate interrupt stack allows the per-process kernel stacks to be sized based on only the requirements for handling a synchronous trap or system call.

Whether the interrupt is handled on the current per-process stack or on a separate interrupt stack, the interrupt handler cannot use that stack to save state between invocations. An interrupt handler must obtain and save all needed information from its global work queue. Because it does not have a per-process context, it cannot relinquish the processor to wait for resources, and must instead run to completion and return.

Software Interrupts

Many events in the kernel are driven by hardware interrupts. For high-speed devices such as network controllers, these interrupts occur at a high priority. For a network controller, it is important to acknowledge receipt of a packet and quickly to reenable the controller to accept another packet, to avoid losing closely spaced packets. However, the further processing of passing the packet to its destination process, although often time consuming, need not be done quickly. Thus, a lower priority is desirable for the further processing, so that critical operations are not blocked from executing for too long.

The mechanism for doing lower-priority processing is called a *software-interrupt process*. Typically, a high-priority interrupt creates a queue of work to be done at a lower-priority level. After queueing a work request, the high-priority interrupt posts a request for a function to be run at the lower-priority level. When the machine priority drops below the lower priority, an interrupt is generated that calls the requested function. If a higher-priority interrupt comes in while the function is executing, the function is preempted while the high-priority event occurs.

The delivery of network packets to destination processes is handled by a packet-processing function that runs at low priority. As packets come in, they are put onto a work queue and the controller is reenabled. Between packet arrivals, the packet-receiving function works to deliver the packets. Thus, the controller can accept new packets without having to wait for the previous packet to be delivered. In addition to network processing, software interrupts are used to handle time-related events and process rescheduling.

3.4 Clock Interrupts

The system is driven by a clock that interrupts at regular intervals. Each interrupt is referred to as a *tick*. On the VAX, the clock ticks 100 times per second. These ticks are used to force context switches so that all processes can get a share of the

ing. This to handle e is a sin- curs, the interrupt. o be sized n call.

c or on a save state ded infor- cess con- instead run

igh-speed rity. For acket and ng closely o its desti- y. Thus, a ations are

software- of work to gh-priority el. When erated that while the rity event

dled by a n, they are et arrivals, controller acket to be ed to han-

h interrupt ind. These hare of the

CPU. At each tick, the system collects statistics about what is happening at that instant, which can be used to do accounting, to monitor what the system is doing, and to determine future scheduling priorities. These interrupts are also used to drive timers both within the system and externally in user processes.

Interrupts for clock ticks are posted at a high hardware-interrupt priority. After the process state has been saved, the *hardclock()* routine is called. It is important that the *hardclock()* routine finish its job quickly.

- If *hardclock()* runs for more than 0.01 second, it will miss its next interrupt.
- Since *hardclock()* maintains the time of day for the system, lost ticks mean that the system will slowly lose track of time.
- Because of the high interrupt priority, while *hardclock()* is running, nearly all other activity on the system is blocked. This blocking can cause network controllers to miss packets, or a disk controller to miss the transfer of a sector coming under a disk drive's head.

To minimize the time spent in *hardclock()*, time-consuming but less time critical processing is handled by a lower-priority software-interrupt handler called *softclock()*.

The work done by *hardclock()* is as follows:

- Charge the currently running process with a tick; if the process has accumulated four ticks, recalculate its priority. If it has a real-time timer or a profiling timer (see Section 3.5), decrement the timer and post a signal if expired. If the current process is over its CPU time limit, post a SIGXCPU signal to the process.
- Increment the time-of-day clock.
- Collect statistics on what the system was doing at the time of the tick (idle, executing in user mode, or executing in system mode).

The remaining time-related processing involves processing timeout requests and periodically reprioritizing processes that are ready to run. These functions are handled by the *softclock()* routine. When *hardclock()* finishes, it checks to see whether there are any *softclock()* functions to be done. If there are, it schedules a *softclock* interrupt. As an optimization, *hardclock()* checks to see whether the state of the processor is such that the *softclock()* interrupt will occur as soon as the *hardclock* interrupt returns. If so, *hardclock()* explicitly lowers the processor priority and calls *softclock()* directly, thus avoiding the cost of returning from one interrupt and immediately reentering another interrupt. Because interrupts are expensive, and because these interrupts occur 100 times per second, the savings add up substantially over time.

Timeouts

The primary task of the *softclock()* routine is to arrange for the execution of periodic events, such as

- Network retransmission timers
- Watchdog timers on peripherals that require constant monitoring
- System-scheduling events

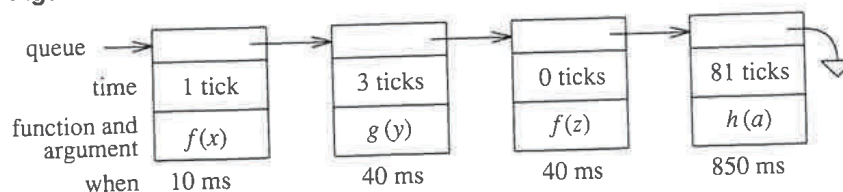
The data structure that describes the set of events is called the *callout queue*. Figure 3.2 shows a snapshot of the callout queue. When a process wishes to request a future event, it specifies a function to be called, an argument to be passed to the function when the event occurs, and the number of clock ticks in the future that the event should occur.

The queue is sorted in the order that the events are to occur; events to happen soonest are at the front, and the most distant events are at the end. The time for each event is kept as a difference from the time of the previous event on the queue. Thus, the *hardclock()* routine needs only to decrement the time to expire of the first element of the queue to decrement time for all events. The *softclock()* routine removes events from the front of the queue whose time has decremented to zero until it finds an event with a still-future (positive) time. Events are typically posted much less frequently than 100 times per second that ticks must be processed and the queue checked to see whether any events are to occur. Thus, it is more efficient to identify the proper location to place an event when that event is posted than it would be to scan the entire queue to determine which events should occur.

The reason for providing an argument as well as a function to be called is to allow a single function to be used by multiple processes. For example, the real-time timer function may be used by several different processes. There is a single function that receives acknowledgment that a timer has expired and that sends a signal to a process. Each process that has a real-time timer running will post a timeout request for this function. The argument that is passed to the function is a pointer to the process-table entry for the process requesting the timeout. This pointer enables the timeout function to post the signal properly. The alternative would require a unique function name for each process that wished to post a real-time timeout.

Timeout processing is more efficient when the timeouts are specified in ticks. Time updates require only an integer decrement, and checks for timer expiration require only a comparison against zero. If the timers contained time values, the comparing and decrementing would be more complex, at least in requiring the

Figure 3.2 Timer events in the callout queue. Ms—milliseconds.



decrement to be by some constant other than one. When the number of events that must be managed gets large, the cost of linear searching to place new events can dominate the simple linear queue used in 4.3BSD. Other possible approaches include maintaining a heap with the next occurring event at the top, and maintaining separate queues of short-, medium- and long-term events [Barkley & Lee, 1988; Varghese & Lauck, 1987].

Process Scheduling

The clock routine is responsible for accumulating CPU time to processes. In addition, it is responsible for periodically running the scheduler that raises and lowers the CPU priority for each process in the system based on the processes recent CPU usage (see Section 4.4). The rescheduling calculation is done once per second. The scheduler is started at boot time. Each time that it runs, it posts itself onto the timeout queue 1 second in the future.

On a heavily loaded system with many processes, the scheduler may take a long time to complete its job. Its timeout is posted 1 **second** after the completion of the scheduling task. On such a system, the **scheduling** will occur less frequently than once per second. Since the scheduler is not responsible for any time-critical functions, such as maintaining the time of day, this is not a problem.

3.5 Timing

The kernel provides several different timing services to processes. These services include timers that run in real time and timers that run only while a process is executing.

Real Time

The system's notion of the current Universal Coordinated Time (UTC) and the current time zone is returned by the system call *gettimeofday*. Most processors today, including all the VAX processors, maintain a battery-backup time-of-day register. This clock continues to run even if the processor is powered down. When the system boots, it consults the processor's time-of-day register to find out the current time. The system's time is then maintained by the hardclock interrupts. At each interrupt, the system increments its global time variable by an amount equal to the number of microseconds per tick. For the VAX running at 100 ticks per second, each tick represents 10,000 microseconds. If the system loses hardclock interrupts, its internal clock will slow. For this behavior to be avoided, the clock must interrupt at high priority, and must finish its job quickly.

Adjusting the Time

Often, it is desirable to maintain the same time on all the machines on a network. It is also possible to have more accurate time than that available from the basic processor clock. For example, devices are readily available that listen to WWV,

the set of radio stations that broadcast UTC synchronization signals in the United States. A set of processes communicating over the network can arrange to agree on a common time. They then wish to change the clock on their host processor to agree with the networkwide timebase. One alternative is to set the system time to the network time using the *settimeofday* system call. For machines whose clocks were fast, this would result in time running backward. This would confuse user programs (such as *make*) that expect time to be nondecreasing. To avoid this problem, the system provides an *adjtime* system call to adjust the time [Gusella *et al.*, 1986]. The system call takes a time delta either positive or negative, and changes the rate at which time advances by 10 percent, faster or slower, respectively, until the adjustment has been effected. The speedup is done through incrementation of the global time by 11,000 microseconds on each hardclock tick; the slowdown is done through incrementation of the global time by 9000 microseconds on each hardclock tick. In either case, time monotonically increases, so user processes that depend on ordering of file-modification times are not adversely affected. Time changes that take tens of seconds to adjust will affect programs that are measuring time intervals by using repeated calls to *gettimeofday*.

External Representation

The kernel maintains its internal time as microseconds since January 1, 1970, UTC. Time is always exported from the system as microseconds rather than as clock ticks, to provide a resolution-independent format. Internally, the kernel is free to select whatever tick rate is desirable to trade off clock overhead with timer resolution. As the tick rate per second increases, the time spent in fielding hardclock interrupts increases. However, the higher the tick rate, the finer the resolution of the system timers. As processors become faster, the tick rate can be increased to provide finer resolution without adversely affecting any existing user applications.

The system has a compiled-in value that specifies in what time zone the machine resides, and a magic number specifying whether and what algorithm to use to calculate daylight saving time. Internally, all timestamps are maintained in UTC. Conversion to local time is handled externally to the system in the C library. There is a library routine that requests the time zone and daylight-saving-time algorithm number from the kernel. This C library routine is responsible for converting kernel timestamps according to local conventions.

There are problems with traditional UNIX time-zone handling, including that in 4.3BSD. Because the rules are compiled into the C library, they are unchangeable on binary distributions, and are difficult to change in source distributions. The compiled-in ruleset is not complete or accurate for many places where UNIX systems are used. In 4.3BSD, it is also not possible for a user to set the desired time zone per process.

The Olson time-zone-handling method (see Section 1.2) that CSRG first released in the 4.3BSD Tahoe release avoids these problems by putting the rules in database files and allowing individual processes to select a set of rules by

specifying a ruleset. This method is extremely flexible, being able to account for multiple rule changes in the same year (e.g., the proposed U.S. presidential election time, by which the West Coast moves ahead 3 hours for 1 day only), for offsets of more than 1 hour (double daylight saving time, as used in World War II) or less (as in Newfoundland), and even for solar time (as in Saudi Arabia). Offsets are still done in seconds, however, as even solar time does not require higher precision. Missing rules can be added at the destination, as can later real-world changes to time-zone rules (which happen frequently and with little notice).

Interval Time

The system provides each process with three interval timers. The *real* timer decrements in real time. It could be used by a library routine to maintain a wakeup-service queue. A SIGALRM signal is delivered when this timer expires. As discussed in Section 3.1, the real-time timer is run from the timeout queue maintained by the *softclock()* routine.

Profiling

The *profiling* timer decrements both in process virtual time (when running in user mode) and when the system is running for the process. It is designed to be used by processes to profile their execution statistically. A SIGPROF signal is delivered when the profiling timer expires. The profiling timer is implemented by the *hardclock()* routine. Each time it is run, it checks to see whether the currently running process has requested a profiling timer; if so, *hardclock()* increments the timer, and, if the timer interval is reached, sends the process a signal.

The *virtual* timer decrements in process virtual time. It runs only when the process is executing in user mode. A SIGVTALRM signal is delivered when it expires. The virtual timer is implemented in *hardclock()* exactly as the profiling timer is, except that it increments the timer for the current process only if it is executing in user mode, not if it is running in the kernel.

Normally, all time-oriented events in the system run off the same clock. Thus, statistics gathering occurs at the same time as rescheduling. This can lead to anomalous statistics, particularly for processes that are driven off system-generated timer events. To allow the collection of more accurate statistics, the kernel supports a second profiling clock. When a second clock is available, it is set to run at a tick rate that is relatively prime to the main system clock (109 ticks per second on the VAX). The statistics gathering is then done by the alternate clock, whereas the timing and scheduling are still done by the original clock.

3.6 Process Management

The memory organization and layout associated with a UNIX process is shown in Fig. 3.3. Each process begins execution with three memory segments, called text, data, and stack. The data segment is divided into initialized data and uninitialized

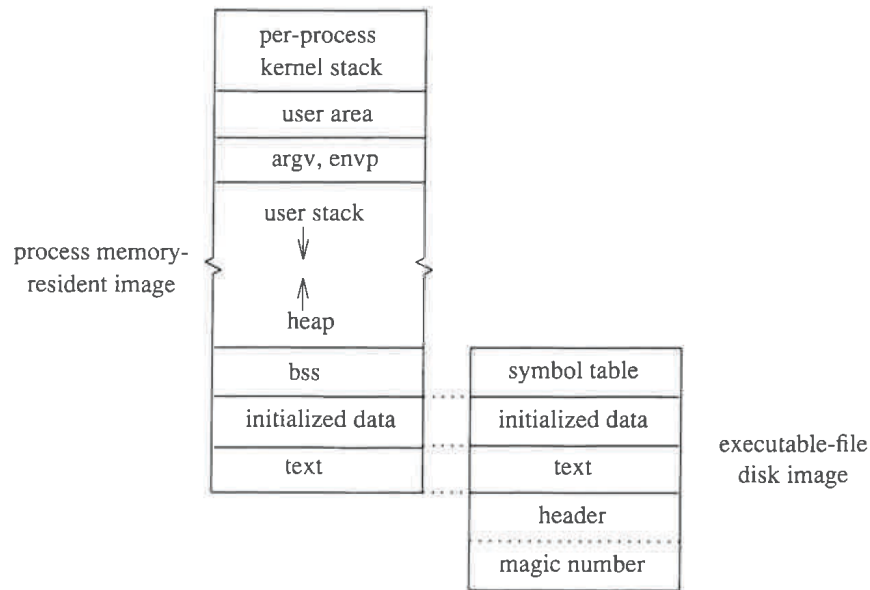


Figure 3.3 Layout of a UNIX process in memory and on disk.

data (bss). The text is read-only and is shared while the data and stack areas are private to the process.

The text and initialized data for the process are taken from the file named when the process is executed. An executable file is distinguished by its being a plain file (not a directory, special file, or symbolic link) and by its having one or more execute bits set. The first few bytes of the file (the first four bytes on the VAX) contain a *magic number* that specifies what type of executable file that file is. Executable files fall into two major classes:

1. Files that are directly executable
2. Files that must be executed by an interpreter

For interpreted files, the magic number is the two-character sequence `#!` followed by a string specifying the pathname of the interpreter to be used. For example, `#!/bin/sh` might refer to the Bourne shell. The kernel executes the named program, passing the name of the file that is to be interpreted as an argument. To prevent loops, only one level of interpretation is allowed; a file's interpreter may not itself be interpreted.

Other files are directly executable. The magic number of such a file specifies whether it can be paged and whether the text part of the file can be shared among multiple processes. Following the magic number is an *exec* header that specifies the sizes of text, initialized data, uninitialized data, and some additional information for debugging. Following the header is an image of the text followed by an

image of the initialized data. Uninitialized data are not contained in the executable file, as they can be created with zero-filled memory. The debugging information is not used by the kernel or by the executing program.

To begin execution, the kernel arranges to have the text portion of the file mapped into the low part of the process address space. The initialized data portion of the file is mapped into the address space following the text. An area equal to the uninitialized data region is created with zero-filled memory after the initialized data region. The stack is also created from zero-filled memory. Although the stack should not need to be zero filled, early UNIX systems made it so. In an attempt to save some startup time, the kernel was modified to skip the zero filling of the stack, leaving the random previous contents of the page instead. Numerous programs stopped working because they depended on the local variables in their *main* procedure being initialized to zero. Consequently, the zero filling of the stack was restored.

Copying the entire text and initialized data portion of a large program into memory causes a long startup latency. We can avoid this startup time by *demand paging the program into memory rather than preloading it*. In demand paging, the program is loaded in small pieces (pages) as it is needed, rather than all at once before it begins execution. The system does demand paging by dividing up the address space into equal-sized pages. For each page, the kernel records the offset into the executable file of the corresponding data. The first access to an address on each page causes a page-fault trap into the kernel. The page-fault handler reads the correct page of the executable file into the process memory. Thus, the kernel loads only those parts of the executable file that are really needed. Chapter 5 explains paging details.

The uninitialized data area may be extended with zero-filled pages through use of the system call *sbrk*, although most user processes use the library routine *malloc()* for this purpose (*malloc()* calls *sbrk*). This newly allocated memory, starting from the top of the original data segment, is called the *heap*. On the VAX, the stack grows down from the top of memory, whereas the heap grows up from the bottom of memory.

Above the user stack is an area of memory that is created by the system when the process is started. Directly above the user stack is the process environment vector (*envp*) and the argument vector (*argv*) set up when the program was executed. At the top of user memory is the user area (*u.*) and the per-process kernel stack. In 4.3BSD, the user address space also contains the user-mode execution state of a process. The execution state is saved on each entry into the kernel, and is restored on each exit. The user area is visible at the top of the user address space because it is used in the implementation of signal handling on the VAX; ports of 4BSD to other hardware may not have this arrangement.

In addition to the information maintained in the user area, a process also requires the use of some global system resources. The kernel maintains a *process table* that contains an entry for each process created in the system. Among other things, the process table records information on scheduling and on virtual-memory allocation. Because the entire process address space, including the user

area, may be swapped out of main memory, the process table must record enough information to be able to locate the user area and to bring the latter back into memory. The scheduling information is maintained in the process table rather than in the user area to avoid swapping in the process only to decide that it is not at a high-enough priority to be run. Similarly, information about events that the process wants to ignore are stored in the process table.

Other global resources associated with a process include space in tables to record information about descriptors, page tables that record information about physical-memory utilization, and space in the text table that coordinates the sharing of common texts.

3.7 User and Group Identifiers

There are a number of objects in a UNIX system that must be identifiable so that the kernel can implement access-control mechanisms. The most important of these objects is a *user*. Users are identified by a 16-bit number termed a *user identifier* (UID). UIDs are not assigned by the kernel—they are assigned by an outside administrative authority. UIDs are the basis for accounting, for controlling the use of disk space, for accessing the filesystem, and for restricting access to sundry privileged kernel operations, such as the request used to reboot a running system. A single user, termed the *superuser* (also known by the user name *root*), is trusted by the system and is permitted to do any kernel operation it requests (with certain exceptions—even the superuser cannot write to a directory or link to an existing file). The superuser has a UID of zero.

Users are organized into *groups*. Each group is assigned a 16-bit number called a *group identifier* (GID), presumably by the same authority that assigns UIDs. GIDs, like UIDs, are used in the access-control facilities provided by the filesystem. UIDs are also used in deciding to what processes a signal may be sent, and in accounting. GIDs are normally used only in determining filesystem-access permissions.

Every UNIX process is associated with a user. The state of the process includes a UID and a set of GIDs. A process's filesystem-access privileges are completely defined by the UID and GIDs of the process (for the filesystem hierarchy beginning at the process's root directory). Normally, these identifiers are inherited automatically from the parent process when a new process is created. Only the superuser is permitted to alter the UID and GID of a process. This scheme by itself enforces a strict compartmentalization of privileges and ensures that no user other than the superuser can *gain* privileges.

Groups may be used in file-access validation. Each file has three sets of permission bits, for read, write, or execute permission for each of user, group, and other. These are checked in order:

1. If the UID of the file is the same as the UID of the process, only the owner permissions apply; the group and other permissions are not checked.

2. If the UIDs do not match, but the GID of the file matches one of the GIDs of the process, the group permissions must allow the requested operation. The other permissions are not checked.
3. Only if the UID and GIDs of the process fail to match those of the file are the permissions for all others checked. These permissions must allow the requested operation.

The UID and GIDs for a process are inherited from that process's parent. When a user logs in, the login program (see Section 13.4) sets the UID and GIDs before performing the *exec* system call to run the user's login shell. All processes run from the login shell therefore inherit the appropriate identifiers. The UID and GIDs are maintained in the user area.

GIDs are implemented as one distinguished GID and a supplementary array of GIDs. The supplementary array is of a fixed size (16 in 4.3BSD), that is settable at the time the system is compiled. Two separate data structures were used to avoid changing existing source code; logically, they are treated as one set of GIDs.

Often, it is desirable to grant a user certain privileges in a constrained manner. For example, a user who wants to send mail must be able to append the mail to another user's mailbox. Making the target mailbox writable by all users would permit a user other than its owner to destroy messages in it (whether maliciously or unintentionally). To solve this problem, the kernel allows the creation of programs that are granted additional privileges when they are run. Programs that run with an additional group privilege are called *set-group-identifier* (*setgid*) programs; programs that run with a different UID are called *set-user-identifier* (*setuid*) programs [Ritchie, 1979]. When a *setuid* program is executed, the permissions of the process are augmented to include those of the UID associated with the program. The UID of the program is termed the *effective UID* of the process, whereas the original UID of the process is termed the *real UID*. Similarly, executing a *setgid* program augments a process's permissions with those of the program's GID and the *effective GID* and *real GID* are defined accordingly.

Setuid and *setgid* programs can be used, for example, to provide controlled access to files or services. The program that delivers mail runs with the privileges of the superuser that can write to any file in the system. Thus, users need not be given the privilege to write other users' mailboxes, but can still do so by running this program, which has been carefully written to append data to mailboxes without colliding with other mail deliveries and without deleting any existing messages.

The *setgid* capability is implemented by addition of the group of the file to the supplementary groups of the process that executed the *setgid* program. Permissions can then be checked as with a normal process. Because of the additional group, the *setgid* program usually will be able to access more files than can a user process that runs a program without the special privilege.

The *setuid* capability is implemented by the *effective UID* of the process being changed from that of the user to that of the program being executed. As

with `setgid`, the protection mechanism will now permit access without any change or special knowledge that the program is running `setuid`. Since a process can have only a single UID at a time, it may lose some privilege in running `setuid`. The previous UID is still held as the real UID when the new effective UID is installed. The real UID, however, is not used for any validation checking. Processes that need to use both the original UID privilege and the `setuid` privilege can ask the system to swap the real and effective UIDs. The protection mechanism uses whichever identifier is currently installed as the effective UID.

Host Identifier

Two additional identifiers are defined by the kernel for use on machines operating in a networked environment. A 32-bit *host identifier* and a string specifying a host's name are maintained by the kernel. These values are intended to be defined uniquely for each machine in a network. Use of the host identifier permits applications easily to craft networkwide unique identifiers for objects such as processes, files, and users, by concatenating the local identifier for the object with the host identifier. The ability to generate networkwide unique identifiers can be useful in the construction of distributed applications [Gifford, 1981]. The host identifier and host name for a machine are defined outside the kernel.

3.8 Resource Controls

All systems have limits imposed by their hardware architecture and configuration; to ensure reasonable operation, these limits must not be exceeded. At a minimum, the hardware limits must be imposed on processes that run on the system. Sometimes, it may be desirable to limit processes further, below these hardware-imposed limits. The system measures resource utilization and allows limits to be imposed on consumption either at or below the hardware-imposed limits.

Process Priorities

The system gives CPU scheduling priority to processes that have not used CPU time recently. This priority scheme tends to favor interactive processes and processes that execute for only short periods. The priority selected for each process is maintained internally by the kernel. The calculation of the priority is affected by the per-process *nice* variable. Positive *nice* values mean that the process is willing to receive less than its share of the processor. Negative values of *nice* mean that the process should be given more than its share of the processor. Most processes run with the default *nice* of zero which gives neither higher nor lower access to the processor. It is possible to determine or change the *nice* currently assigned to a process, to a process group, or to the processes of a specified user. There are many other factors besides the *nice* that affect scheduling. These include the amount of CPU time that the process has used recently, the amount of memory that the

process has used recently, and the current load on the system. The exact algorithms that are used are described in Section 4.4.

Resource Utilization

As a process executes, it uses system resources, such as the CPU and memory. The kernel tracks the resources used by each process and compiles statistics describing this usage. The statistics managed by the kernel are available to a process while the latter is executing; when the process terminates, the statistics are made available to its parent.

The resources used by a process are returned by the system call *getrusage*. The resources used by the current process, or by all the terminated children of the current process, may be requested. This information includes

- The amount of user and system time used by the process
- The memory utilization of the process
- The paging and disk I/O activity of the process
- The number of voluntary and involuntary context switches taken by the process
- The amount of interprocess communication done by the process

The resource-usage information is collected at locations throughout the kernel. The CPU time is collected at each tick in *hardclock()*. The kernel calculates memory utilization by sampling the amount of memory an active process is using each second. The scheduler does memory sampling at the same time that it is recomputing process priorities. The *pagein()* routine recalculates the paging activity each time it starts a disk transfer to fulfill a paging request (see Section 5.11). The I/O activity statistics are collected each time the process has to start a transfer to fulfill a file or device I/O request. The IPC communication activity is counted each time a packet is sent through the socket interface.

Resource Limits

The kernel controls the usage of certain per-process resources. These controls include

- The maximum amount of CPU time that can be accumulated
- The maximum size of a file that can be created
- The maximum size of a process's data segment
- The maximum size of a process's stack segment
- The maximum size of a core file
- The maximum amount of physical memory a process may use at any given moment.

For each resource controlled by the kernel, two limits are maintained: a *soft limit* and a *hard limit*. Only the superuser can raise a hard limit. Other users may alter a soft limit within the range from 0 to the corresponding hard limit, or (irreversibly) may only lower the hard limit. If a process exceeds some soft limits, the kernel delivers a signal to the process to notify it that a resource limit has been exceeded. Normally, this signal causes the process to terminate, but the process may elect to catch or ignore the signal.

The limits are enforced at the same locations that the resource statistics are collected. The CPU time limit is enforced in *hardclock()*. The physical-memory limit is enforced by *pageout()*. While a process exceeds its limit for physical memory, the *pageout()* routine will take any page that it finds away from the process. The stack and data-segment limits are enforced by a return of allocation failure once those limits have been reached. The file-size limit is enforced by the filesystem. Once the file-size limit is reached, the process is sent a signal. The default action is to kill the process. If the process catches or ignores the signal and attempts to continue writing, it will get error returns.

Filesystem Quotas

In addition to limits on the size of individual files, the kernel will also enforce limits on the total amount of space that a user can utilize on a filesystem. Our discussion of the implementation of these limits is deferred to Section 7.6.

3.9 System Operation

There are several operational functions having to do with system startup and shutdown. The bootstrapping operations are described in Section 13.2. System shutdown is described in Section 13.5.

Accounting

The system supports a simple form of resource accounting. As each process terminates, an accounting record describing the resources used by that process is written to a systemwide accounting file. The information supplied by the system comprises

- The name of the command that ran
- The amount of user and system CPU time that was used
- The elapsed time the command ran
- The average amount of memory used
- The number of disk I/O operations done
- The UID and GID of the process
- The terminal from which the process was started

oft limit
ay alter
eversi-
he ker-
is been
process

tics are
memory
physical
he pro-
ocation
l by the
al. The
e signal

nce lim-
discus-

nd shut-
m shut-

cess ter-
ocess is
system

The information in the accounting record is drawn from the run-time statistics that were described in Section 3.8. The granularity of the time fields is in sixty-fourths of a second. To conserve space in the accounting file, the times are stored in a 16-bit word as a *floating-point* number using three bits as a base-8 exponent, and the other 13 bits as the fractional part. For historic reasons, the same floating-point conversion routine processes the count of disk operations. Thus, the number of disk operations is multiplied by 64 before it is converted to the floating-point representation.

There are also Boolean flags that tell how the process terminated, whether it used superuser privileges, and whether it used *exec* after *fork*.

The superuser requests accounting by passing a name for the accounting file to the kernel. As part of terminating a process, the kernel appends an accounting record to the accounting file. The kernel makes no use of the accounting records; the records' summaries and usage are entirely the domain of user-level accounting programs. As a guard against running out of space on a filesystem due to unchecked growth of the accounting file, the system suspends accounting when the filesystem is reduced to only 2 percent free space. Accounting resumes when the filesystem has at least 4 percent free space.

The accounting information has certain limitations. The information on run time and memory usage is only approximate, because it is gathered statistically. Accounting information is written only when a process exits, so processes that are still running when a system is shut down unexpectedly do not show up in the accounting file. These unaccounted processes include all the long-lived system daemons. Finally, the accounting records fail to include much information that is needed to do accurate billing. The missing information includes usage of other resources, such as tape drives, printers, and network communication.

Exercises

- 3.1 Describe three types of system activity.
- 3.2 When can a routine executing in the top half of the kernel be preempted? When can it be interrupted?
- 3.3 Why are routines executing in the bottom half of the kernel precluded from using information located in the user area?
- 3.4 Why does the system defer as much work as possible from high-priority interrupts to lower-priority software-interrupt processes?
- 3.5 What determines the shortest (nonzero) time period that a user process can request when setting a timer?
- 3.6 How does the kernel determine the system call for which it has been invoked?

- 3.7 How are initialized data represented in an executable file? How are uninitialized data represented in an executable file? Why are the representations different?
- 3.8 Why is the conversion from UTC to local time done by user processes, rather than in the kernel?
- 3.9 Describe the security implications of not zero filling the stack region at program startup.
- 3.10 Describe how the “#!” mechanism can be used to make programs that require emulation appear as though they were normal executables.
- 3.11 Is it possible for a file to have permissions set such that its owner cannot read it, even though a group can? Is this possible if the owner is a member of the group that can read the file?
- *3.12 What is the advantage of having the kernel, rather than an application, restart an interrupted system call?
- *3.13 Describe a scenario in which the sorted-difference algorithm used for the callout queue does not work well. Suggest an alternative data structure that runs more quickly than does the sorted difference algorithm for your scenario.
- *3.14 The SIGPROF profiling timer was originally intended to replace the *profil* system call to collect a statistical sampling of a program's program counter. Give two reasons why the *profil* facility had to be retained.
- *3.15 Should a setuid program that swaps its effective and real UIDs be allowed to swap them back again? Why or why not?
- **3.16 What weakness in the process-accounting mechanism makes the latter unsuitable for use in a commercial environment?

References

- Barkley & Lee, 1988.
 - R. E. Barkley & T. P. Lee, “A Heap-Based Callout Implementation to Meet Real-Time Needs,” *USENIX Association Conference Proceedings*, pp. 213–222 (June 1988).
- Gifford, 1981.
 - D. Gifford, “Information Storage in a Decentralized Computer System,” PhD Thesis, Electrical Engineering Department, Stanford University, Stanford, CA (1981).
- Gusella *et al.*, 1986.
 - R. Gusella, S. Zatti, & J. M. Bloom, “The Berkeley UNIX Time Synchronization Protocol,” pp. 22:1–10 in *UNIX System Manager's Manual*, 4.3

Berkeley Software Distribution, Virtual VAX-11 Version, USENIX Association, Berkeley, CA (1986).

Ritchie, 1979.

D. M. Ritchie, "Protection of Data File Contents," *United States Patent* (4,135,240), United States Patent Office (January 16, 1979). Assignee: Bell Telephone Laboratories, Inc., Murray Hill, NJ, Appl. No.: 377,591, Filed: Jul. 9, 1973.

Varghese & Lauck, 1987.

G. Varghese & T. Lauck, "Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility," *Proceedings of the Eleventh Symposium on Operating Systems Principles*, pp. 25-38 (November 1987).

PART 2

Processes

Process Management

4.1 Introduction

A *process* is a program in execution. A process must have system resources, such as memory and the underlying CPU. UNIX supports the illusion of concurrent execution of multiple processes by scheduling system resources among the set of processes that are ready to execute. In this chapter, we describe the composition of a process, the method the system uses to switch between processes, and the scheduling policy used to promote sharing of the CPU. In later sections, we study process creation and termination, the signal facilities, and process-debugging facilities.

Two months after the beginning of 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 a new program in 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 modified to permit several processes to remain in memory simultaneously, in order to reduce swapping. But this modification 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].

A UNIX process operates in either *user mode* or *kernel mode*. In user mode, a process executes application code with the machine in a nonprivileged protection mode. When a process requests services from the operating system with a system

call, it switches into the machine's privileged protection mode and operates in kernel mode.

The resources used by a process are similarly 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 processor-status register, and the stack-related registers, as well as the contents of the memory segments that constitute the UNIX notion of a program (the text, data, and stack segments).

Kernel-mode resources include those required by the underlying hardware—such as registers, program counter, and stack pointer—but also by the state required for the UNIX kernel to perform system services for a process. This *kernel state* includes parameters to the current system call, the current process's user identity, scheduling information, and so on. UNIX divides the kernel state for each process into two separate data structures: the *proc structure* and the *user structure*.

The *proc structure* contains information that must always remain resident in main memory, whereas the *user structure* contains information that needs to be resident only when the process is executing (although user structures of other processes also may be resident). User structures are allocated dynamically through the memory-management facilities. The user structure of the currently running process is mapped to a fixed location in the kernel's virtual address space, so that kernel software may refer to it as a static C language structure. Proc structures are allocated from a fixed-sized table created by the kernel at system startup. The *proc structure* for the current process is referenced through a pointer in the current process's user structure.

Multiprogramming

UNIX supports transparent multiprogramming—the illusion of concurrent execution of multiple processes or programs. This is done by *context switching*; that is, by switching between the execution context of processes. A mechanism is also provided for *scheduling* the execution of processes; 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 the process, but the software state used by the kernel must be switched as well.

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 executing user applications. Since most of the work of a context switch is expended in saving and restoring the operating context of a process, reducing the amount of the information required for that context is an effective way of producing faster context switches. Many UNIX systems avoid copying the context states of resident processes by using memory-management hardware (if available) to map the user structure of the new running process into the user structure's

expected location. This mapping could also be done in software by using an indirect structure reference, but that approach would be less efficient.

Scheduling

Fair scheduling of processes is an involved task that is dependent on the types of executable programs and on scheduling policy goals. 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 it takes for a program to complete. A process's priority is periodically recalculated based on parameters such as the amount of CPU time it has used, the amount of memory resources it holds or requires for execution, and so on.

UNIX 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 each process a high execution priority and allows that process to execute for a fixed *time slice*. Processes that execute for the duration of their slice have their priority lowered, whereas processes that give up the CPU (usually because they do I/O) are allowed to remain at their priority. Processes that are inactive have their priority raised. Thus, 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 will get a lower priority, but it will return to a high priority when it is inactive again.

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 executing application code in user mode.

The system must both detect and eliminate thrashing. It detects thrashing by observing the amount of free memory and the rate of memory requests. When the system has few free memory pages and a high rate of new memory requests, it considers itself to be thrashing. The system eliminates thrashing by finding the four largest processes and swapping out the one that has been resident in memory the longest. The memory freed by swapping out the process can then be distributed to the remaining processes, which usually can then proceed. This selection criterion is chosen to favor small **interactive** jobs over large batch jobs. If the thrashing continues, additional **processes** are selected for swapping out until enough memory becomes available for other processes to run effectively. Eventually, enough processes complete and free their memory that swapped-out **processes can return to main memory to run**. However, **even if there is not enough memory**, the **swapped-out processes** are allowed to **return after about 20 seconds**. Usually, the **thrashing condition will return**. But the **swapped-in process**

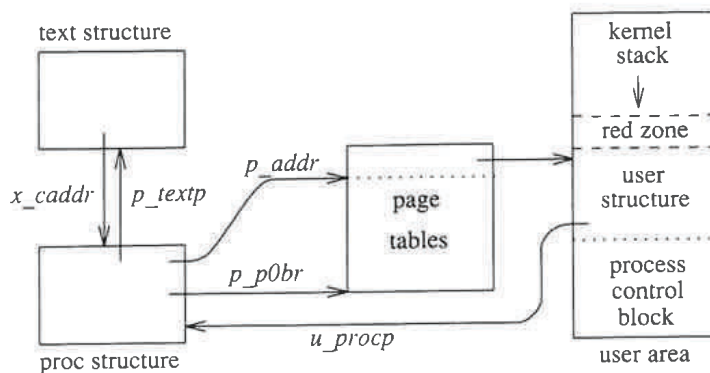


Figure 4.1 Process state.

will no longer have the longest memory-resident time, so some other process will be selected for swapping out.

The orientation of the scheduling policy toward an interactive job mix reflects the original design of UNIX for use in a time-sharing environment. Numerous papers have been written about alternative scheduling policies, such as those used in batch-processing environments or real-time systems. Usually, these policies require changes to the system in addition to alteration of the scheduling policy.

Section 4.2 details the state of a process, including the *proc* and *user* structures. Section 4.3 describes the facilities for context switching and for synchronizing access to system data structures used by both the top and bottom halves of the kernel. Section 4.4 explains the implementation of the priority-based scheduling algorithm described previously. Section 4.5 covers process creation, and Section 4.6 describes process termination.

4.2 Process State

There are several data structures that describe the state of a process, as shown in Fig. 4.1. Some of the structures must remain resident in main memory, whereas others may be moved to secondary storage when memory resources are low. The *proc structure* and *text structure* are the two data structures that must always remain in main memory. The *user structure* and process page tables are needed only when a process is resident and running. The following sections describe the portions of these structures that are relevant to process management. The *text structure* and *page tables* are described more fully in Chapter 5.

The Proc Structure

A *proc structure* contains the process state that must remain in memory at all times, as shown in Table 4.1. This state is logically divided into several categories of information having to do with the following:

Table 4.1 Process data structure.

Category	Name	Description
scheduling	<i>p_pri</i>	current priority; negative is high
	<i>p_usrpri</i>	user priority based on <i>p_cpu</i> and <i>p_nice</i>
	<i>p_nice</i>	user-requested scheduling priority
	<i>p_cpu</i>	recent CPU usage
	<i>p_slptime</i>	amount of time sleeping
identifiers	<i>p_pid</i>	unique process identifier
	<i>p_ppid</i>	parent-process identifier
	<i>p_uid</i>	user identifier
memory management	<i>p_textp</i>	pointer to description of executable file
	<i>p_0br</i>	pointer to process page tables
	<i>p_szpt</i>	size of process page tables
	<i>p_addr</i>	location of user area page tables
	<i>p_swaddr</i>	location of user area when swapped out
synchronization	<i>p_wchan</i>	event process is awaiting
signals	<i>p_sig</i>	mask of signals pending delivery
	<i>p_sigignore</i>	mask of signals being ignored
	<i>p_sigcatch</i>	mask of signals being caught
	<i>p_pgrp</i>	process-group identifier
resource accounting	<i>p_rusage</i>	pointer to <i>rusage</i> structure
	<i>p_quota</i>	pointer to disk-quota structure
timer management	<i>p_time</i>	amount of real time until timer expires

- **Scheduling.** The process priority, user-mode scheduling priority, recent CPU utilization, and amount of time spent sleeping
- **Identification.** The process identifier, the parent-process identifier, and the real user identifier
- **Memory management.** A pointer to a text-table entry used in managing shared text segments, a pointer to the page tables for the process, the size of the set of pages currently resident in main memory, and an address for the user structure and page tables when they are swapped to secondary storage
- **Synchronization.** The *wait channel*, a description of the event for which the process is waiting (see the next section)
- **Signals.** Signals pending delivery and the action to take when a signal is posted to a process, as well as the process group
- **Resource accounting.** The *rusage* structure that describes the utilization of the many resources provided by the system (see Section 3.8), and a pointer to any disk quotas

Table 4.2 Process states.

State	Description
SSLEEP	awaiting an event
SRUN	runnable
SIDL	intermediate state in process creation
SZOMB	intermediate state in process termination
SSTOP	process stopped or being traced

• **Timer management.** The time until a real-time timer is to expire

A process's state has a value, as shown in Table 4.2. When a process is first created with a *fork* system call, it is initially marked as SIDL. The state is changed to SRUN when enough resources are allocated to the process for the latter to begin execution. From that point onward, a process's state will fluctuate between SRUN (runnable), SSLEEP (waiting for an event), and SSTOP (stopped by a signal or the parent process), until the process terminates. A deceased process is marked as SZOMB until its termination status is communicated to its parent process.

The system organizes proc structures into three lists to speed lookups based on process state. These lists are threaded by the *p_next* field of the proc structure. Process-table entries that are in use are either on the *zombproc* list if the process is in the SZOMB state, or on the *allproc* list. Process-table entries that are not in use are found on the *freeproc* list.

Most processes, except the currently executing process, are also in one of two queues: a *run queue* or a *sleep queue*. Processes that are in a runnable state are placed on a run queue, whereas processes that are blocked awaiting an event are located on a sleep queue. Stopped processes not also awaiting an event are on neither type of queue. Both queues are organized as an array of doubly linked lists, using the *p_link* and *p_rlink* fields in the proc structure. The run queues are organized according to process-scheduling priority, and described in Section 4.4. The sleep queues are organized in a hashed data structure that optimizes finding a sleeping process by the event number (wait channel) for which the process is waiting. The sleep queues are described in Section 4.3.

Flags in the *p_flag* field of the proc structure show additional information about a process's *operational* state, as shown in Table 4.3. Most of these flags are used by the kernel to mark transient states during operations that may involve one or more context switches. Other flags are used to record information communicated by a user (SSEQL, SUANOM, SOUSIG) or deduced by the system (SSYS). We shall describe the purpose of each flag as we encounter it in later sections (see Section 13.4 for SLOGIN).

Every process in the system is assigned a unique identifier termed the *process identifier*, or *PID*. PIDs are the common mechanism used by applications and by

Table 4.3 Process flags.

Flag	Description
SLOAD	process loaded in main memory
SSYS	process created by system (e.g., swapper or page daemon)
SLOCK	process is being swapped out
SSWAP	alternate return after swap
STRC	process is being debugged
SWTED	another debugging flag
SOUSIG	process using old signal mechanism
SULOCK	user requested that process not be swapped
SPAGE	process waiting for page to be retrieved
SKEEP	kernel requires process in memory
SOWEUPC	process collecting system-call timing information
SOMASK	restore old mask after taking signal
SWEXIT	process is exiting
SPHYSIO	process doing physical I/O
SVFORK	process resulted from <i>vfork()</i>
SNVM	child has parent's virtual memory during a <i>vfork()</i>
SVFDONE	child returned virtual memory to parent during a <i>vfork()</i>
SPAGI	page in data space from filesystem on demand
SSEQ	user advised of sequential virtual-memory behavior
SUANOM	user advised of random virtual-memory behavior
STIMO	timing out during sleep
SSEL	process selecting; clear if wakeup occurs
SLOGIN	a login process
SPTECHG	page tables for process have changed

the kernel to reference processes. PIDs are used by applications when sending 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 process's own PID, and the PID of the process's parent process.

The *p_pgrp* field and related links (*p_pptr*, *p_cptr*, *p_osptr*, and *p_ysptr*) are used in locating descendent processes, as shown in Fig. 4.2. When a process spawns a child process, a forward link is created from the parent to the child in the *p_cptr* field, and a backward link from the child to the parent is kept in the *p_pptr* field. If a process has more than one child process active at a time, the children are linked together, with the oldest child process forward-linked to its younger sibling processes through *p_ysptr* and backward-linked through *p_osptr*. In Fig. 4.2, process B is a direct descendent of process A, whereas processes C, D, and E are descendents of process B and are siblings of one another. Process B typically

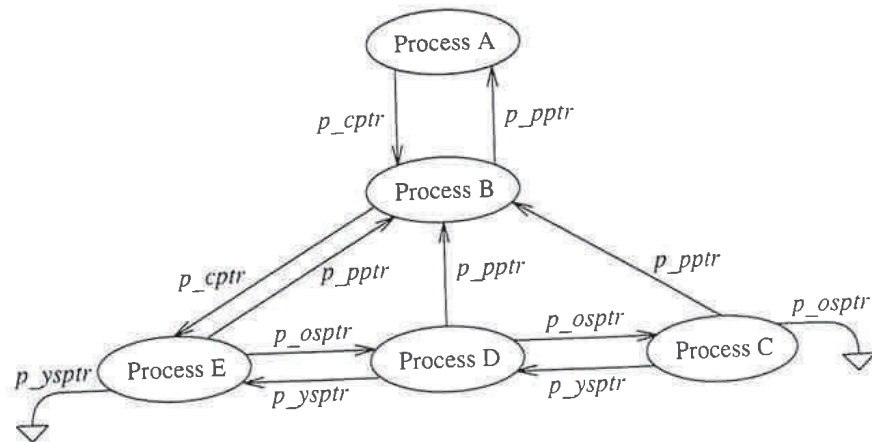


Figure 4.2 Process-group hierarchy.

would be a shell that started a pipeline (see Sections 2.4 and 2.6), including processes C, D, and E. Process A probably would be the system-initialization process *init* (see Sections 3.1 and 13.4).

A *process group* is a collection of related processes, such as a pipeline, that have been assigned the same *process-group identifier*. Process groups are an integral part of the signal facilities. The *killpg* system call can be used to *broadcast* a signal to all the processes in a process group. In Fig. 4.2, processes C, D, and E usually would be placed in a process group separate from that of processes A and B. This separation would allow the job represented by processes C, D, and E to be started, stopped, brought into the foreground, or put into the background as a group. The terminal I/O system (described in Chapter 9) synchronizes access to a terminal by permitting only a single process group to be associated with a terminal at any time. This synchronization facility, combined with the process-group and signal mechanisms, provides the foundation for the job-control facilities described in Section 4.7.

CPU time is made available to processes according to their *scheduling priority*. A process has two scheduling priorities; one for scheduling user-mode execution, and one for scheduling kernel-mode execution. The *p_usrpri* field in the *proc* structure contains the user-mode scheduling priority, whereas the *p_pri* field holds the current kernel-mode scheduling priority. The current priority may be different from the user-mode priority when the process is executing in kernel mode. Priorities range between 0 and 127, with a lower value interpreted as a higher priority (see Table 4.4). User-mode priorities range from PUSER (50) to 127; priorities less than PUSER are used only when a process is *asleep*—that is, awaiting an event in the kernel—and immediately after such a process is awakened. A kernel process that is asleep with a priority in the range PZERO to PUSER may be awakened by a signal. But a process asleep at a priority below PZERO will never be awakened by a signal; this rule is necessary to prevent signals from

Table 4.4 Process-scheduling priorities.

Priority	Value	Description
PSWP	0	priority while swapping process
PINOD	10	priority while waiting for file control information
PRIBIO	20	priority while waiting on disk I/O completion
PZERO	25	baseline priority
PWAIT	30	base priority for waiting on resources
PLOCK	35	base priority for waiting on resource locks
PSLEP	40	base priority for waiting for an event
PUSER	50	base priority for user-mode execution

interfering with operations such as disk transfers that must be completed atomically.

The ordering of the priorities has remained unchanged since the Sixth Edition. All kernel priorities must be better than those assigned to user processes. Because priorities encode whether a system call may be aborted by a signal, the assignment of values above or below PZERO is important. The use of priorities to specify whether a system call can be interrupted by a signal was implemented to save text space in early UNIX kernels. Instead of adding an additional parameter to the *sleep()* routine, which is called in many places in the kernel, the value was encoded in the priority.

The User Structure

The *user structure*, (also called the *u-dot (u.) structure* because all references in C are of the form *u.*), contains the process state that may be swapped to secondary storage. This state includes

- The user- and kernel-mode execution states
- The state related to system calls
- The descriptor table
- The accounting information
- The resource controls
- The per-process execution stack for the kernel

For the VAX, the current execution state of a process is encapsulated in a VAX *process control block (PCB)*. This structure, shown in Fig. 4.3, is defined by the VAX machine architecture and includes the general-purpose registers, stack pointers,

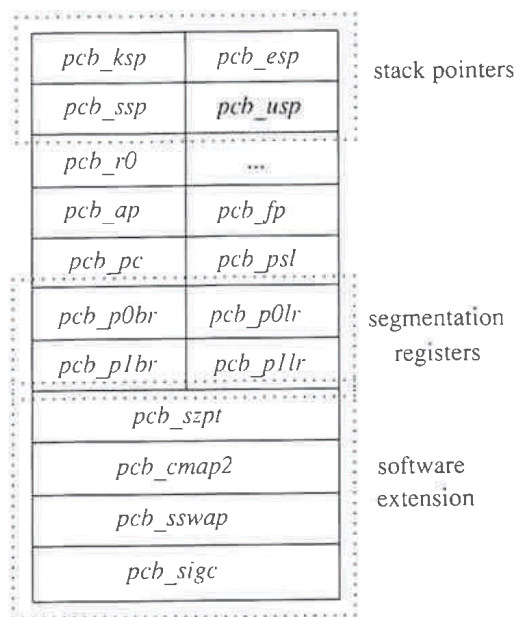


Figure 4.3 VAX process control block (PCB).

program counter, processor status longword, and segment base and length registers. In addition to the information required by the hardware, four fields that are manipulated by the kernel have been added. These fields specify the size of the process page table, two pointers used in context switching and segment copying, and a chunk of code used in implementing the delivery of signals.

The user structure is mapped to a fixed location in the VAX virtual-address space and is marked readable in user mode. Having the user structure readable in user mode permits signal delivery to be implemented more efficiently (see Section 4.7).

The location of the kernel stack in the user structure simplifies context switching by localizing all a process's kernel-mode state in a single structure. This design, however, restricts the stack to a fixed size and requires implementors to be careful when writing code that executes in the kernel. That is, kernel code must avoid using large local variables and deeply nested subroutine calls to avoid overflowing the run-time stack. In the 4.3BSD system for the VAX, 10 pages of 512 bytes each are allocated for each user structure. The kernel stack grows down from the top of the user structure. The fixed-size portion of the user structure, including the PCB, is about 2300 bytes. The page above the end of the fixed-size portion is marked invalid. Thus, overflowing the kernel stack will cause a kernel-access fault instead of overwriting the fixed-sized portion of the user structure, which would cause disastrous results. These uses leave about 2050 bytes for the kernel's run-time stack. Because interrupt processing takes place on a separate *interrupt stack*, the size of the kernel stack in the user structure restricts only that code executed as a result of traps and system calls.

Memory

The memory used by a process is described by fields in the *proc* and *user* structures. A process's user-mode execution state is described by the information in the *PCB* and in the *page tables* that describe which pages of the process's memory are currently resident in main memory. Before a process can execute, the process's page tables must be loaded into memory and the appropriate *VAX* memory-management registers must be set to point at them. For the purposes of discussing process management, we shall ignore most of the details of memory management and assume that memory resources are available. Chapter 5 discusses how memory is managed by the system and details memory-related actions that are required to perform a context switch.

The Text Structure

4.3BSD supports the sharing of memory that holds *pure* code; that is, code that cannot be modified. Each of these memory regions, called a *shared text segment*, is described by a *text structure*. When a program image is loaded with an *execve* system call, if the code segment is marked as *pure*, the system will load the text segment of the image separately from the initialized and uninitialized data segments and will create a text structure that describes the loaded image. (The image may not be brought completely into memory; instead, it may be paged into memory on demand.) Future executions of the same image may then share the text segment by referencing the resources described in the text structure. Process creation and context switching of shared texts are discussed in this chapter; memory management of shared texts is presented in Chapter 5.

4.3 Context Switching

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

Each type of context switching is performed 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 *switch()* and *resume()* routines. Asynchronous event handling is managed by the underlying hardware and is effectively transparent to the system. Our discussion of asynchronous event handling shall focus on how it relates to synchronizing access to kernel data structures.

Process State

Context switching between processes requires that both the kernel- and user-mode context be changed; to facilitate this, the system ensures that all of a process's user-mode state is located in one data structure: the user structure (some 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. Thus, the kernel's hardware-execution state is defined by the contents of the PCB that is located at the beginning of the user structure.
- **User-mode hardware-execution state.** When execution is in kernel mode, the user-mode state of a process (such as copies of program counter, stack pointer, and general registers) always resides on the kernel's execution stack that is located in the user 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 the kernel is entered (see Section 3.1).
- **The proc structure.** The proc structure always remains resident in memory; thus, a pointer to the structure can be held safely in the user structure.
- **The text structure.** Likewise, any shared text used by the process is defined by an entry in the text table that is also a permanently resident data structure. This permits a pointer to the text structure to be held in the proc structure.
- **Memory resources.** Finally, memory resources of a process are effectively described by the contents of the base and length registers located in the PCB and by the values present in the proc structure. As long as the process remains in memory, these values will remain valid, and context switches can be performed without the associated page tables being saved and restored. After a process is swapped to secondary storage, however, these values need to be recalculated when the process returns to main memory.

Low-Level Context Switching

The localization of the context of a process in its user structure permits the kernel to do context switching simply by switching the notion of the current user structure. To make switching user structures as efficient as possible on the VAX, the system maps the physical memory associated with each process's user structure to a fixed location at the top of the user-mode virtual-address space. (This location is also at a fixed virtual address in the kernel's address space.) Since the user-

mode virtual-memory mapping is defined by values maintained in a process's PCB, switching between processes, and between user structures, simply becomes a matter of swapping PCBs. The VAX architecture includes two instructions and a privileged register specifically designed for swapping PCBs. The *Process Control Block Base* (PCBB) register contains the physical address of the PCB for the current process. The *svpctx* instruction *saves* the current machine state in the current PCB, whereas the *ldpctx* instruction *loads* the current machine state from the current PCB. Swapping between two processes is then performed with

<i>svpctx</i>	# save current state in current PCB
...	# load address of a new PCB, and...
<i>ldpctx</i>	# ... establish state from the PCB

These low-level details are hidden inside the *swtch()* and *resume()* routines. Whenever a context switch is required, a call to the *swtch()* routine causes the highest-priority process to run. *Swtch()* first selects the appropriate process from the scheduling queues, then invokes *resume()* with the information needed to load the PCBB register. Once *resume()* has performed a *ldpctx* instruction to restore the execution state of the new process, it must also check the state of the new process for a nonlocal return request (such as when a process first starts execution after a *fork*; see Section 4.5).

Voluntary Context Switching

A *voluntary* context switch occurs whenever a process must await the availability of a resource or the arrival of an event. Voluntary context switches happen frequently in normal system operation. For example, a process typically blocks each time it requests data from a slow input device, such as a terminal. In UNIX, voluntary context switches are initiated through the *sleep()* routine. When a process no longer needs the CPU, it invokes *sleep()* with a *wait channel* and a scheduling priority. The wait channel is typically the address of some data structure that identifies the resource or event for which the process is waiting. Some conventional addresses used for wait channels are

<i>lbolt</i>	wait for up to 1 second
<i>proc</i>	wait for a child process to terminate
<i>u</i>	wait for a signal
<i>buf</i>	wait for a block I/O operation to complete

The priority specified in a *sleep()* call is the priority that should be assigned to the process when that process is awakened. This priority does not affect the user-level scheduling priority. Sleeping processes are organized in an array of queues

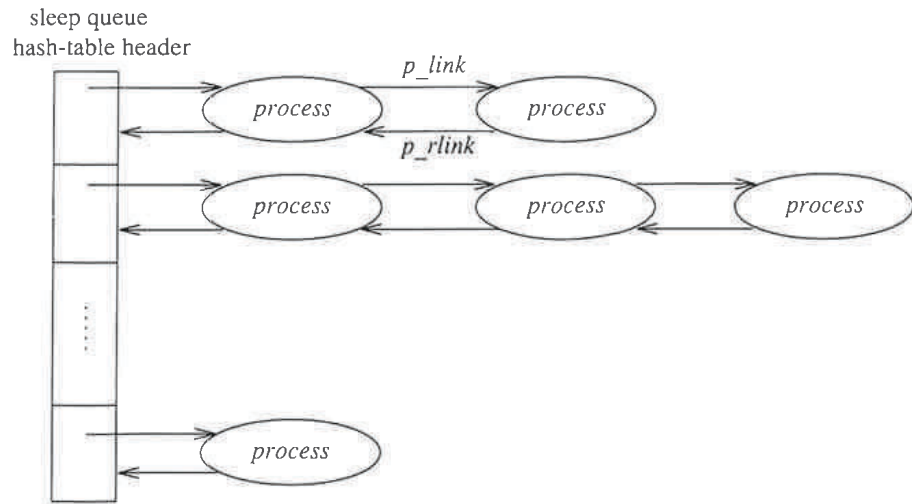


Figure 4.4 Queueing structure for sleeping processes.

(see Fig. 4.4). The *sleep()* and *wakeup()* routines hash wait channels to calculate an index into the sleep queues.

The *sleep()* routine performs the following steps in its operation:

1. Prevent interrupts that might cause process-state transitions by raising the hardware-processor priority level to *splhigh*
2. Record the wait channel in the proc structure and hash the wait-channel value to locate a sleep queue for the process
3. Set the process's priority to the priority the process will have when the process is awakened
4. Place the process at the *end* of the sleep queue selected in step 2
5. Call *switch()* to request that a new process be scheduled; the hardware priority level is implicitly reset as part of switching to the other process

A sleeping process is not selected to execute until it is removed from a sleep queue and marked runnable. This operation is performed by the *wakeup()* routine. A process that puts itself to sleep with a priority between PZERO and PUSER is interruptible; that is, it may be awakened and marked runnable if a signal is posted to it. For example, a process waiting for input from a terminal will wait at an interruptible priority. Otherwise, a sleeping process is uninterruptible; that is, it will not be scheduled to run until the event for which it is waiting occurs. For example, a process waiting for disk I/O will sleep at an uninterruptible priority.

Processes usually sleep at uninterruptible priorities so that they can hold kernel resources while they wait. Processes that sleep at interruptible priorities 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 processes must release any resources that they hold before going to sleep. Because of the extra work that must be done to prepare for an interruptible sleep, the only time that these sleeps are used is when a process must wait for events that may not occur for a long time. Occasionally, an event that is supposed to occur quickly, such as a tape I/O, will get held up because of a hardware failure. Because the process is sleeping in the kernel at an uninterruptible priority, it will be impervious to any attempts to send it a signal, even one that should cause it to exit unconditionally.

The *wakeup()* routine is called to signal that an event has occurred or that a resource is available. *Wakeup()* is invoked with a wait channel, and it awakens *all* processes sleeping on that wait channel. A *wakeup()* operation processes entries on a sleep queue from *front* to *back*. For each process that needs to be awakened, *wakeup()*

1. Removes the process from the sleep queue
2. Recomputes the user-mode scheduling priority if the process has been sleeping longer than 1 second
3. Makes the process runnable if it is in a SSLEEP state and places it on the run queue if it is not swapped out of main memory; if the process has been swapped out, the *swapon* process will be awakened to load it back into memory (see Section 5.13); if the process is in a SSTOP state, it is left on the queue until it is explicitly restarted by a user-level process, either by a *ptrace* system call or by a *continue* signal (see Section 4.7)

If *wakeup()* moved any processes to the run queue and one of them had a higher scheduling priority than does the currently executing process, it will also request that the CPU be rescheduled as soon as possible.

The most common use of *sleep()* and *wakeup()* is in scheduling access to shared data structures; this use is described in the section on *synchronization*, later in this chapter.

Intraprocess Context Switching

In addition to the interprocess context-switching mechanism, a mechanism is also provided for performing *nonlocal gotos*. This facility permits a routine to transfer control to a calling routine at some place *up* the execution stack, as shown in Fig. 4.5. Nonlocal gotos normally are used to handle an exceptional condition that should abort an operation in progress. For example, if a signal is received while the kernel is processing a system call, the kernel may abort the system call and return control to the system-call handler where an error code can be returned to the application.

Nonlocal gotos are implemented with the *setjmp()* and *longjmp()* routines, which work analogously to the C library routines *setjmp()* and *longjmp()*. The *setjmp()* routine saves the minimal amount of information about the current stack

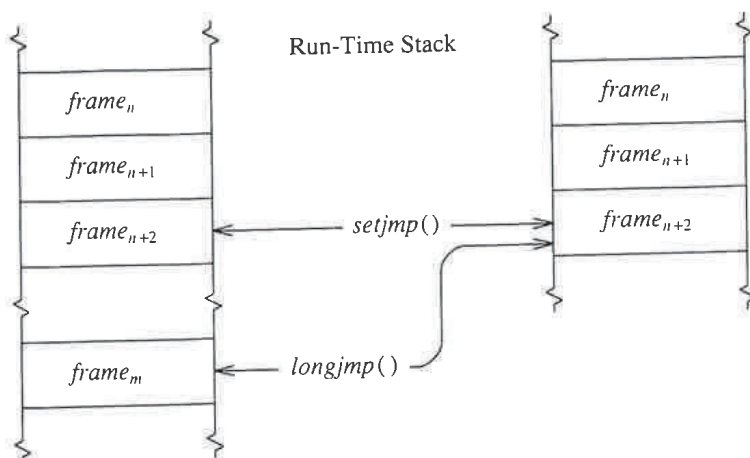


Figure 4.5 Stack-frame management for a nonlocal goto.

frame to allow a later *longjmp()* call to unwind the execution stack to the point at which the *setjmp()* call was taken. When *setjmp()* is first called, its return value is zero. A call to *longjmp()* causes *setjmp()* to return a second time with a nonzero value. The state saved by a call to *setjmp()* typically includes the kernel-mode frame pointer and program counter. When a process is swapping out itself, the last operation that it does is to request that its u-dot area (including its kernel stack) be written out to backing store. The I/O operation may begin (from a disk interrupt) before the process has had a chance to call the *sleep()* routine. Thus, it is necessary to save the contents of all the general-purpose registers before making the I/O request, so that the process can be started successfully when it is swapped back in from secondary storage. The *savectx()* and *ldctx()* routines are used for these situations.

Nonlocal gotos are used sparingly in the system and are never nested. Two entries in the user structure, *u_qsave* and *u_ssave*, are used to hold the state required to do nonlocal gotos when a signal interrupts a system call and when execution context is being restored after being swapped in from secondary storage, respectively. A more general, nestable, exception-handling facility would be useful in reducing the complexity of some parts of the system.

Synchronization

Interprocess synchronization to a resource typically is implemented by the association with the resource of two flags; a *locked* flag and a *wanted* flag. When a process wants to access a resource, it first checks the locked flag. If the resource is not currently in use by another process, this flag should not be set, and the process can simply set the locked flag and use the resource. If the resource is in use, however, the process should set the wanted flag and call *sleep()* with a wait channel associated with the resource (typically the address of the data structure used to describe the resource). When a process no longer needs the resource, it clears the

Table 4.5 VAX interrupt-priority assignments.

Name	Priority	Blocks
<i>spl0()</i>	0	nothing (normal operating mode)
<i>splsoftclock()</i>	8	low-priority clock processing
<i>splnet()</i>	12	network protocol processing
<i>spl4()</i>	20	low-priority devices
<i>spltty()</i>	21	terminal multiplexers
<i>splbio()</i>	21	disk and tape controllers
<i>splimp()</i>	22	network device controllers
<i>splclock()</i>	24	high-priority clock processing
<i>splhigh()</i>	31	all interrupt activity

locked flag and, if the wanted flag is set, invokes *wakeup()* to awaken all the processes that called *sleep()* to await access to the resource.

Routines that run in the bottom half of the kernel do not have a context and consequently cannot wait for a resource to become available by calling *sleep()*. When the top half of the kernel accesses resources that are shared with the bottom half of the kernel, it cannot use the locked flag to ensure exclusive use. Instead, it must prevent the bottom half from running while it is using the resource. Synchronizing access with routines that execute in the bottom half of the kernel requires knowledge about when these routines may run. Although machine-dependent, most implementations of UNIX order interrupt priorities according to Table 4.5. To block interrupt routines at and below a certain priority level, an appropriate *set-priority-level* call must be made. For example, when a process needs to manipulate a terminal's data queue, the code that accesses the queue is written in the following style:

```
s = spltty();    /* raise priority to block tty processing */
...             /* manipulate tty */
splx(s);        /* reset priority level to previous value */
```

Processes must take care to avoid deadlocks when locking multiple resources. Suppose two processes, A and B, require two resources, R_1 and R_2 , to perform some operation. If process A acquires R_1 and process B acquires R_2 , then a deadlock occurs when process A tries to acquire R_2 and process B tries to acquire R_1 . Since a UNIX process executing in kernel mode is never preempted by another process, locking multiple resources is relatively simple, although it must be done **carefully**. If a process knows that multiple resources are required to perform an **operation**, then it can safely lock one or more of these resources in any order, as long as it never voluntarily relinquishes control of the CPU. If, however, a process cannot acquire all the resources it needs, it must release any resources it

holds before calling *sleep()* to wait for the currently inaccessible resource to become available.

Alternatively, if resources can be partially ordered, it is necessary only that they be allocated in an increasing order. For example, as the *namei()* routine traverses the filesystem name space, it must lock the next component of a path-name before it relinquishes the current component. A partial ordering of path-name components exists from the root of the name space to the leaves. Thus, translations down the name tree can request a lock on the next component without concern for deadlock. However, when it is traversing up the name tree (i.e., following a pathname component of dot-dot (..)), the kernel must take care to avoid sleeping while holding any locks.

Raising the processor priority level to guard against interrupt activity works for a uniprocessor architecture, but not for a shared-memory multiprocessor machine. Similarly, much of the UNIX kernel implicitly assumes that kernel processing will never be done concurrently. Numerous vendors—such as Sequent, AT&T, Encore, and Alliant—have redesigned the synchronization schemes and eliminated the uniprocessor assumptions implicit in the standard UNIX kernel, so that UNIX will run on tightly coupled multiprocessor architectures.

4.4 Process Scheduling

4.3BSD uses a process-scheduling algorithm based on *multilevel feedback queues*. All processes that are runnable are assigned a scheduling priority that determines in which *run queue* they are placed. In selecting a new process to run, the system scans the run queues from highest to lowest priority and chooses the first process on the first nonempty queue. If multiple processes 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 process blocks, it is not put back onto any run queue. If a process 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 process 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 overall system throughput, because the system will have less overhead from doing context switches, and processor caches will be flushed less often. The time quantum used by 4.3BSD is one-tenth of a 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.

The system adjusts the priority of a process dynamically to reflect resource requirements (e.g., being blocked awaiting an event) and the amount of resources consumed by the process (e.g., CPU time). Processes are moved between run queues based on changes in their scheduling priority (hence, the word *feedback* in the name *multilevel feedback queue*). When a process other than the currently running process attains a higher priority, the system switches to that process when

the time quantum for the first process expires; if a sleeping process is awakened, the current process is preempted after any current system call. The system tailors this *short-term scheduling algorithm* to favor interactive jobs by raising the scheduling priority of processes that are blocked for a *long time* waiting for I/O, and by lowering the priority of processes that accumulate significant amounts of CPU time.

Short-term process scheduling is broken up into two parts. The next section describes when and how a process's scheduling priority is altered; the section after describes the management of the run queues and the interaction between process scheduling and context switching.

Calculations of Process Priority

A process's scheduling priority is directly determined by two values contained in the *proc* structure: *p_cpu* and *p_nice*. The value of *p_cpu* provides an estimate of the recent CPU utilization of the process. The value of *p_nice* is a user-settable weighting factor that ranges numerically between -20 and 20. The normal value for *p_nice* is 0. Negative values increase a process's priority, whereas positive values decrease its priority.

A process's user-mode scheduling priority is calculated every four clock ticks (40 milliseconds on a VAX) by this equation:

$$p_usrpri = PUSER + \left[\frac{p_cpu}{4} \right] + 2 \cdot p_nice \quad (\text{Eq. 4.1})$$

Values must be at least PUSER (see Table 4.4); values greater than 127 are clamped to 127. This calculation causes the priority to decrease linearly based on recent CPU utilization. The user-controllable *p_nice* parameter acts as a limited weighting factor. Negative values retard the effect of heavy CPU utilization by offsetting the additive term containing *p_cpu*. Otherwise, if one ignores the second term, *p_nice* simply shifts the priority by a constant factor.

The CPU utilization, *p_cpu*, is incremented each time the system clock ticks and the process is found to be executing. In addition, *p_cpu* is adjusted once per second using a digital decay filter. The decay causes about 90 percent of the CPU usage accumulated in a 1-second interval to be forgotten over a period of time that is dependent on the system *load average*. To be exact, *p_cpu* is adjusted according to

$$p_cpu = \frac{(2 \cdot \text{load})}{(2 \cdot \text{load} + 1)} p_cpu + p_nice \quad (\text{Eq. 4.2})$$

where the *load* is a sampled average of the length of the run queue over the previous 1-minute interval of system operation.

To understand the effect of the decay filter, we can consider the case where a single compute-bound process monopolizes the CPU. The process's CPU utilization will accumulate clock ticks at a rate dependent on the clock frequency. The load average will be effectively 1, resulting in a decay of

$$p_cpu = 0.66 \cdot p_cpu + p_nice$$

If we assume the process accumulates T_i clock ticks over time interval i , and that p_nice is zero, then the CPU utilization for each time interval will count into the current value of p_cpu according to

$$\begin{aligned} p_cpu &= 0.66 \cdot T_0 \\ p_cpu &= 0.66 \cdot (T_1 + 0.66 \cdot T_0) = 0.66 \cdot T_1 + 0.44 \cdot T_0 \\ p_cpu &= 0.66 \cdot T_2 + 0.44 \cdot T_1 + 0.30 \cdot T_0 \\ p_cpu &= 0.66 \cdot T_3 + \cdots + 0.20 \cdot T_0 \\ p_cpu &= 0.66 \cdot T_4 + \cdots + 0.13 \cdot T_0 \end{aligned}$$

Thus, after five decay calculations, only 13 percent of T_0 remains present in the current CPU utilization value for the process. Since the decay filter is applied once per second, another way of saying this is that about 90 percent of the CPU utilization is forgotten after 5 seconds.

Processes that are runnable have their priority adjusted periodically as just described. However, the system ignores processes blocked awaiting an event: These processes cannot accumulate CPU usage, so an estimate of their filtered CPU usage can be calculated in one step. This optimization can significantly reduce a system's scheduling overhead when many blocked processes are present. The system recomputes a process's priority when that process is awakened and has been sleeping for longer than 1 second. The system maintains a value, $p_slptime$, that is an estimate of the time a process has spent blocked waiting for an event. The value of $p_slptime$ is set to 0 when a process calls *sleep()*, and is incremented once per second while the process remains in an SSLEEP or SSTOP state. When the process is awakened, the system computes the value of p_cpu according to

$$p_cpu = \left[\frac{(2 \cdot load)}{(2 \cdot load + 1)} \right]^{p_slptime} p_cpu \quad (\text{Eq. 4.3})$$

and then recalculates the scheduling priority using Eq. 4.1. This analysis ignores the influence of p_nice ; also, the *load* used is the current load average, not the load average at the time the process blocked.

Process-Priority Routines

The priority calculations used in the short-term scheduling algorithm are spread out in several areas of the system. Two routines, *schedcpu()* and *roundrobin()*, run periodically. *Schedcpu()* recomputes process priorities once per second, using Eq. 4.2, and updates the value of $p_slptime$ for processes blocked by a call to *sleep()*. The *roundrobin()* routine runs 10 times per second and causes the system to reschedule the processes in the highest-priority (nonempty) queue in a round-robin fashion, and allows each process a 100-millisecond time quantum.

The CPU usage estimates are updated in the system clock-processing module, *hardclock()*, which executes 100 times per second. Each time a process accumulates four ticks in its CPU usage estimate, p_cpu , the system recalculates the priority of the process. This recalculation uses Eq. 4.1 and is done by the *setpri()* routine. The decision to recalculate after four ticks is related to the management of

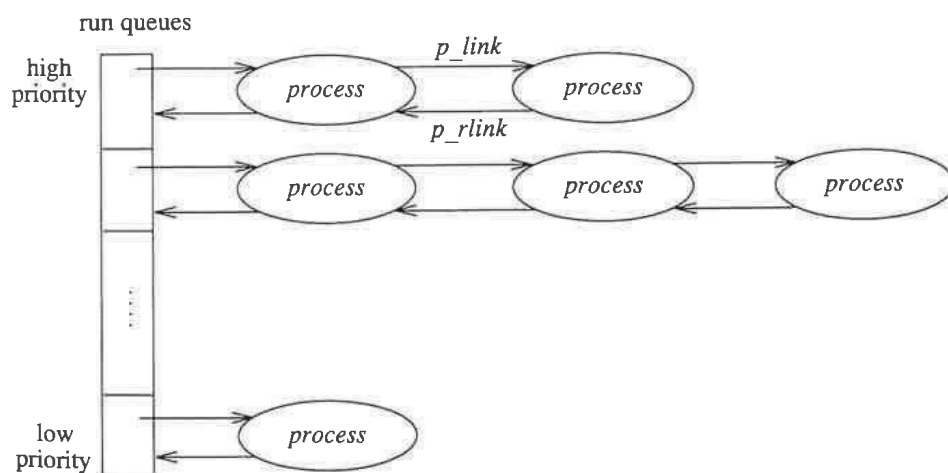
the run queues described in the next section. In addition to the call from *hardclock()*, each time *setrun()* places a process on a run queue, it also calls *setpri()* to recompute the process's scheduling priority. This call from *wakeup()* to *setrun()* operates on a process other than the currently running process. Accordingly, *wakeup()* invokes *updatepri()* to recalculate the CPU usage estimate according to Eq. 4.3 before calling *setpri()*.

Process Run Queues and Context Switching

The scheduling-priority calculations are used to order the set of runnable processes. The scheduling priority ranges between 0 and 127, with 0 to 49 reserved for scheduling processes executing in kernel mode, and 50 to 127 reserved for scheduling processes executing in user mode. The number of queues used to hold the collection of runnable processes affects the cost of managing the queues. If only a single (ordered) queue is maintained, then selecting the next runnable process becomes simple, but other operations become expensive. Using 128 different queues can significantly increase the cost of identifying the next process to run. The system uses 32 run queues, selecting a run queue for a process by dividing the process's priority by four. The selection of 32 different queues is a compromise based mainly on the availability of certain VAX machine instructions that permit the lowest-level scheduling algorithm to be implemented efficiently.

The run queues contain all the runnable processes in main memory except the currently running process. Figure 4.6 shows how each queue is organized as a doubly linked list of proc structures. The head of each run queue is kept in an array, *qs[]*, and associated with this array is a bit vector, *whichqs*, that is used in identifying the nonempty run queues. Two routines, *setrq()* and *remrq()*, are used to place a process at the tail of a run queue, and to take a process off the head

Figure 4.6 Queueing structure for runnable processes.



of a run queue. The heart of the scheduling algorithm is the *swtch()* routine. *Swtch()* is responsible for selecting a new process to run; it operates as follows:

1. Look for a nonempty run queue. This is easy to do on the VAX through use of the *ffs* (find first bit set) instruction on the *whichqs* bit vector. If *whichqs* is zero, there are no processes to run and *swtch()* will unblock interrupts and loop; this is the *idle loop*.
2. Given a nonempty run queue, remove the first process on the queue. On the VAX, this can be done with a single instruction, *remque*.
3. If this run queue is now empty as a result of removing the process, reset the appropriate bit in *whichqs*.
4. Reset the *noproc* and *runrun* flags. The *noproc* flag is a global flag used to show that *no process is currently running*. The *runrun* flag shows that a context switch should take place; this is described later.
5. Set the new process running with a call to *resume()*.

On the VAX, the *swtch()* routine is coded in assembly language for efficiency.

Given the *swtch()* routine and the process-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 process calls the *sleep()* routine. *Sleep()* can be invoked by only a runnable process, so it need only place the process on a sleep queue and invoke *swtch()* to schedule the next process to run. The *swtch()* routine, however, cannot be called from code that executes at interrupt level, because the VAX architecture does not permit context switching from the interrupt stack.

An alternative mechanism must exist. This mechanism is to set a global *reschedule request* flag, named *runrun*, and then to post an *asynchronous system trap* (AST) for the current process. An AST is a trap that is delivered to a process the next time that process returns to user mode. The VAX architecture directly supports ASTs in hardware; other systems emulate ASTs by, for example, forcing memory faults. The system interrogates the *runrun* flag and forces a context switch by calling *swtch()* in each of these situations:

- After returning from an interrupt, if the system was idling
- At the end of a system call or trap
- When the AST trap occurs

The *wakeup()*, *setpri()*, *roundrobin()*, *schedcpu()*, and *setrun()* routines all set *runrun* and post an AST to force rescheduling and, possibly, an involuntary context switch.

It should be noted that, because UNIX does not preempt processes executing in kernel mode, real-time response to events is dependent on the amount of time

spent in each system activity. In addition, processes have no way to determine which of their pages are resident, so they have no way of ensuring that they will be able to execute a sequence of instructions without incurring one or more page faults. Since the system guarantees no upper bounds on the duration of a system activity, UNIX is decidedly not a real-time system. Attempts to retrofit UNIX with real-time process scheduling have addressed this problem in different ways [Ferrin & Langridge, 1980; Sanderson *et al.*, 1986].

4.5 Process Creation

In UNIX, new processes are created with the *fork* system call. There is also a *vfork* system call that differs from *fork* in how the virtual-memory resources are treated; *vfork* is described in Chapter 5.

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 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 proc structure for the child process
2. Duplicating the context of the parent (including the user structure and virtual-memory 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 5. Consequently, only those actions related to process management will be described here.

The system maintains a linked list of free process-table entries, so an unused entry can be found by examining the head of the list, *freeproc*.

While creating a new process, the system initializes most of the child's proc structure from that of the parent. One value that is not propagated, however, is the PID, which must be unique among all processes. Early versions of UNIX 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. 4.3BSD maintains a range of unallocated PIDs between *mpid* and *pidchecked*. A new PID is allocated by using the value of *mpid*, and *mpid* is then incremented. When *mpid* 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—zombie and swapped processes also are checked).

The child process inherits all the privileges and limitations of its parent. In addition to the open files and other information contained in the *u-dot* area, the *proc* entry of a child process inherits the following information from the parent:

- The signal state—the UID for delivering signals, process group, and signal actions
- The *p_nice* scheduling parameter
- Program-segment information—the size of each segment and the page tables, and, if a shared text segment is being used, a pointer to the associated text table entry
- The in-core disk-quota information (see Section 7.6)

To duplicate a process's image, the memory-management facilities are invoked through a call to *procdup()*. The *procdup()* routine is passed a pointer to the initialized *proc* structure for the child process and is expected to allocate all the resources that the child will need to execute. A call to *procdup()* returns a value of 1 in the child process and of 0 in the parent process. The kernel returns different values by arranging for a nonlocal goto when the child process is first set running. When *procdup()* is called, it fills in the *pch_sswap* field in the child's PCB, and invokes *savectx()*, passing the address of *u_ssave* in the user structure of the child process:

```
procdup(p, isvfork)
    struct proc *p;
{
    ...                /* allocate and initialize child u */
    childu.u_pcb.pcb_sswap = &childu.u_ssave;
    if (savectx(&childu.u_ssave))
        return (1); /* return 1 in child process */
    ...                /* duplicate process image */
    return (0);
}
```

The *resume()* routine checks the *pch_sswap* field in the software extension to the PCB after each context switch. If this field is nonzero, *resume()* invokes the *ldctx()* routine, passing it the value found in *pch_sswap* as the location for a non-local goto, returning control to the *procdup()* routine at the point where the original *savectx()* call was made, but with an alternate return value. This series of actions permits *procdup()* to return different values to its caller, depending on whether it is executing in the context of the child or parent process.

Finally, the new child process is made known to the scheduler by its being placed on the run queue, and the return value from *procdup()* is passed back to become the return value of the *fork* system call.

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 it still exists). This termination status is returned through the *wait* or *wait3* system calls. The *wait3* call is a variant of the standard *wait* call that permits an application to request the status of both stopped and terminated processes. *Wait3* can also request statistics describing the resource utilization of a terminated child process. Also, unlike *wait*, *wait3* allows a process to request status codes without blocking.

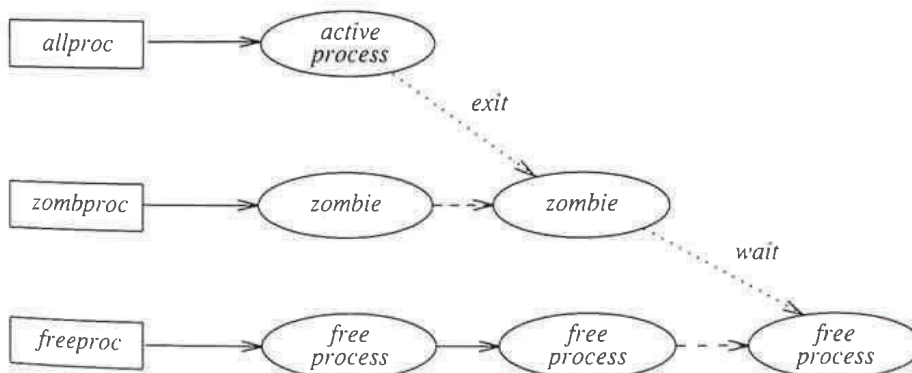
Within the kernel, a process terminates by calling the *exit()* routine. *Exit()* first cleans up the process's kernel mode execution state by

- 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*, as shown in Fig. 4.7. The process state is changed and the global flag *noproc* is marked to show that no process is currently running. The *exit()* routine then

- Records the termination status in the *p_xstat* field of the *proc* 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 *proc* structure

Figure 4.7 Management of the *proc* structure during process termination.



- Notifies the deceased process's parent

Finally, after the parent has been notified, the CPU is rescheduled directly through a call to *swtch()*.

The *wait* and *wait3* calls both work by searching a process's descendant processes for processes that have terminated. If a process in SZOMB state is found, the system will copy the termination status from the deceased process and then reclaim the associated process structure, reinitializing it to a state suitable for future use. In particular, the process-table entry is taken off the zombie list and is returned to the free list pointed to by *freeproc*. Note that resources used by children of a process are accumulated only as a result of a *wait* system call. When trying to analyze the behavior of a long-running program, users would find it useful to be able to obtain this information before the termination of a process.

4.7 Signals

UNIX 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 4.6. Signals may be delivered *asynchronously* to a process through application-specified *signal handlers*, or may result in *default* actions, such as process termination, carried out by the system. 4.3BSD signals are designed to be software equivalents of hardware interrupts or traps.

Each signal has an associated *action* that defines how a signal should be handled when it is delivered to a process. Each signal has a *default* action that may be any one of

- Ignoring the signal
- Terminating the process
- Terminating the process after generating a *core file* that contains the process's execution state at the time the signal was delivered
- Stopping the process

An application program may use the *sigvec* system call to specify an action for a signal, including

- 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, which is

Table 4.6 Signals defined in 4.3BSD.

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
SIGIOT	create core image	I/O trap instruction executed
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
SIGUSR1	terminate process	user-defined signal 1
SIGUSR2	terminate process	user-defined signal 2

thus *caught*. The two signals, SIGSTOP and SIGKILL, may not be ignored or caught; this restriction ensures that a software mechanism exists for stopping and killing runaway processes. It is not possible for a user 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* or *killpg* system calls. A sending process may post signals only to 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 may always be sent to any descendent 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, the delivery of signals may be *masked* by a process. The execution state of each process contains a set of signals currently masked from delivery. If a signal posted to a process is being masked, the signal is recorded in the process's set of pending signals, but no action is taken until the signal is unmasked. The *sigblock* and *sigsetmask* system calls modify a set of masked signals for a process. The *sigblock* call *adds* to the set of masked signals, whereas the *sigsetmask* call *replaces* the set of masked signals.

The system does not allow the SIGKILL, SIGSTOP, or SIGCONT signals to be masked. Preventing SIGCONT from being masked is necessary in 4.3BSD only because the delivery of this signal is required to resume stopped processes. POSIX, for example, specifies alternative semantics for SIGCONT that permit users to mask the signal without causing problems.

Two other signal-related system calls are *sigpause* and *sigstack*. The *sigpause* call permits a process to relinquish the processor until it receives a signal. This facility is similar to the system's *sleep()* routine. The *sigstack* 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 its normal run-time stack. In some applications, however, this is unacceptable. For example, if an application is running on a stack that the system does not automatically expand, and the stack overflows, the signal handler must execute on an alternate stack. This facility is similar to the *interrupt-stack* mechanism provided by VAX hardware.

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

Process Groups

A *process-group identifier* is a 16-bit number used to identify a collection of processes, a *process group*, such as a pipeline set up by the shell. Process-group identifiers are used by the *killpg* and *setpriority* system calls as a means of identifying a collection of processes, and by the terminal handler as an identification mechanism when multiplexing access to a terminal. A process inherits its process group from its parent process. The system never changes the process group of a process that has one; only user programs (usually shells) do this with the *setpgid* system call. For process-group initialization, see Section 13.4.

Within the system, the two main users of process groups are the terminal handler (Chapter 9) and the interprocess-communication facilities (Chapter 10). 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 a terminal. When a process attempts to *read* from or *write* to a terminal, or otherwise to alter the state of a terminal with an *ioctl* call, and the process's process-group identifier is different from the terminal's process-group identifier, the terminal handler will *stop* the process by posting a signal that causes the process to enter an SSTOP state. This facility is the basic mechanism used in implementing *job control*; it is described in more detail later in this chapter.

Comparison with Other Systems

UNIX 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. Thus, reliability became desirable, and there was pressure from the user community for it. Rather than trying to fit the necessary major changes into the existing signal mechanism, the developers designed a completely new framework to contain the old capabilities as a subset while accommodating new mechanisms.

4.3BSD signal facilities differ radically from the facilities found in other versions of UNIX. In particular, 4.3BSD signals address the following problems:

- On older systems, the action associated with a signal is reset to the default action when the signal is received by a process. Thus, a process that wants to handle receipt of multiple instances of a given signal must reinstall the handler after each signal is received. This produces a race condition: Another instance of the signal may be received before the handler is reinstalled, causing the default action to take place, rather than what the programmer intended. In 4.3BSD, signal handlers are permanently installed with a single *sigvec* system call.
- On older systems, the currently handled signal is not masked while it is being handled by an application. This permits a later signal to terminate a process, as described previously. In addition, the signal not being masked means recursive interrupts may occur unless the signal handler marks the signal to be ignored. In 4.3BSD, the signal currently being handled is automatically masked from delivery while an application's signal handler is invoked.
- Previous UNIX systems did not permit user processes to mask the delivery of signals. The only mechanism provided to an application that needed to guard a critical region against signals was the ability to disable delivery of those signals.

Further, because only one signal at a time could be disabled, it was not possible to guarantee that a collection of signals could be disabled before one was delivered to the process. In 4.3BSD, sets of signals can be masked with the *sigblock* and *sigsetmask* system calls. If a signal is posted to a process that has it currently masked, the delivery of the signal is postponed until the signal is unmasked.

- In previous UNIX systems, signals were always delivered on the normal run-time stack of a process. In 4.3BSD, an alternate stack may be specified for delivering signals with the *sigstack* system call. Signal stacks permit programs that manage fixed-sized run-time stacks to handle signals reliably.
- On older UNIX systems, system calls interrupted by a signal caused the call to be terminated prematurely and an EINTR error code to be returned. In 4.3BSD, system calls interrupted by a signal are automatically restarted whenever possible and reasonable. Automatic restarting of system calls permits programs to service signals without having to check the return code from each system call to determine whether the call should be restarted. 4.3BSD (but not 4.2BSD) contains a flag for *sigvec* that will cause the old semantics to apply.

The signal facilities found in 4.3BSD are designed around a UNIX *virtual-machine* model, in which UNIX system calls are considered the parallel of machine's hardware instruction set. Signals are the software equivalent of traps or interrupts, and signal-handling routines perform 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 performed, 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 run-time stack. These machine models are summarized in Table 4.7.

Table 4.7 Comparison of hardware machines and software virtual machines.

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

Changes to 4.3BSD Signals in POSIX

4.3BSD signals have been adopted by POSIX, except for signal stacks. There have been some syntax changes in the signal functions, however, and because of them all the names have been changed, as summarized in Table 4.8. The only function found in POSIX that has no equivalent in 4.3BSD is *sigpending*; this routine determines what signals have been posted but have not yet been delivered.

Posting 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 process. 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 process. But when a signal forces a process to be stopped, the action is instead sometimes carried out 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 process, and the process is then set to run (or is awakened if it was sleeping at an interruptible priority level). The *p_cursig* field in the proc structure, if nonzero, contains the signal currently being delivered to a process. The *p_sig* field contains the set of signals pending delivery to a process. Each time a process returns from a call to *sleep()* (with the process priority less than PZERO), or prepares to exit the system after processing a system call or trap, it checks to see whether a signal is pending delivery. If a signal is pending and must be delivered in the process's context, it is moved from the pending set to *p_cursig*,

Table 4.8 4.3BSD and POSIX signal-handling facilities.

4.3BSD	POSIX
<i>sigmask()</i>	sigsetopts functions
<i>sigblock()</i>	<i>sigprocmask()</i>
<i>sigsetmask()</i>	<i>sigprocmask()</i>
<i>sigpause()</i>	<i>sigsuspend()</i>
<i>sigvec()</i>	<i>sigaction()</i>
<i>sigstack()</i>	(none)
<i>sigreturn()</i>	(none)
(none)	<i>sigpending()</i>
<i>signal()</i>	<i>signal()</i>

and the process invokes the *psig()* routine to carry out 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:

1. Determine the action that the receiving process will take when the signal is delivered. This information is kept in the *p_sigignore*, *p_sigmask*, and *p_sigcatch* fields of the process's proc structure. If a process is not ignoring, masking, 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 may return.
2. Given an action, *psignal()* adds the signal to the set of pending signals, *p_sig*, and then does any implicit actions specific to that signal. For example, if the signal is a *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 process is currently masking delivery of the signal, *psignal()*'s work is complete and it may return.
4. If, however, the signal is not being masked, *psignal()* must either perform the action directly, or arrange for the process to execute so that it will carry out the action associated with the signal. To get the process running, *psignal()* must interrogate the state of the process.

SSLEEP The process is blocked awaiting an event. If the process is sleeping at a negative priority, then nothing further can be done. Otherwise, the action can be applied, either directly, or indirectly by waking up the process. There are two actions that can be applied directly. For signals that cause a process to stop, the process is placed in an SSTOP state, and the parent process is notified of the state change by posting of a SIGCHLD signal to it. For signals that are ignored by default, the signal is removed from *p_sig* and the work is complete. Otherwise, the action associated with the signal must be performed in the context of the receiving process, and the process is placed onto the run queue with a call to *setrun()*.

SSTOP 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 caught). If the signal is SIGCONT, then the process is set running again unless it is blocked

waiting on an event; if the process is blocked, it is returned to SSLEEP state. If the signal is SIGKILL, then the process is set running again no matter what, so that it can terminate the next time it is scheduled to run. Otherwise, the signal causes the process to be made *runnable*, but the process is not placed on the run queue, as it must wait for a continue signal.

SRUN, SIDL, SZOMB

If the process is not the currently executing process, an AST is posted so that the signal will be noticed by the receiving process as soon as possible.

The implementation of *psignal()* is complicated, mostly because *psignal()* controls the process-state transitions that are part of the job-control facilities and because it interacts strongly with process-debugging facilities.

Delivering a Signal

Most actions associated with delivering a signal to a process are carried out within the context of that process. A process checks its *proc* structure for pending signals at least once each time it enters the system, by calling the *issig()* routine.

Issig() takes the first signal it encounters in the set of pending signals, *p_sig*, and places that signal in the *p_cursig* field. If delivering the signal causes a signal handler to be invoked or a core dump to be performed, the caller is notified that a signal is pending, and actual delivery is done by a call to *psig()*. That is,

```
if (issig())
    psig();
```

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

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

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

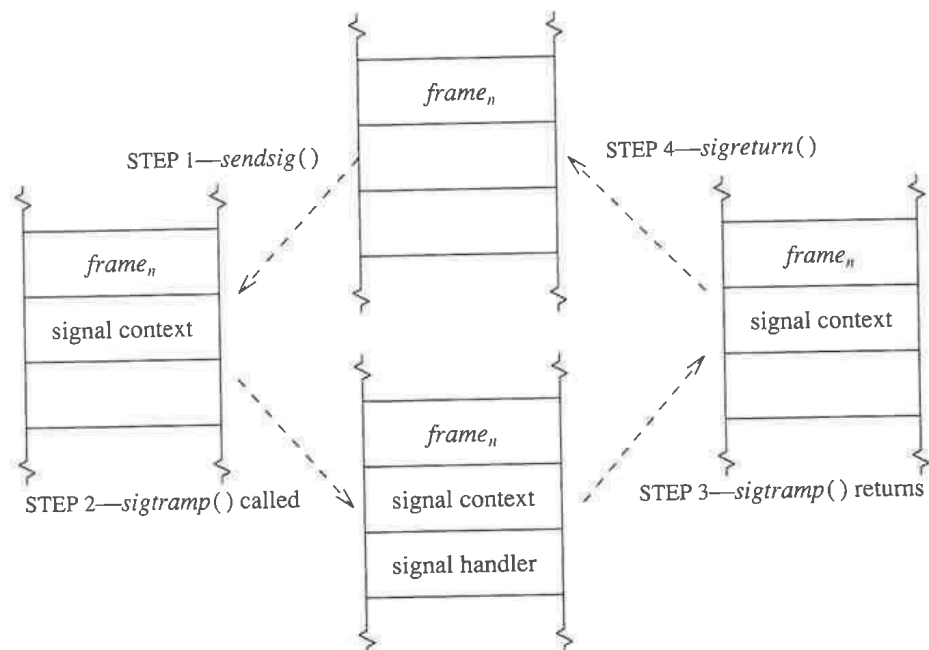
The former task is performed by the *core()* routine and is always followed by a call to *exit()* to force process termination. To invoke a signal handler, *psig()* first calculates a set of masked signals and installs it in *p_sigmask*. This set normally includes the signal being delivered, so that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

The implementation of the *sendsig()* routine is machine-dependent. Figure 4.8 shows the flow of control associated with signal delivery on the VAX. For the VAX, an argument list and the process's current user-mode execution context are stored on the signal stack (possibly different from the normal run-time stack), and the state of the process is manipulated so that, on return to user mode, a call will be immediately made to a body of code stored in the process's user structure. (The u-dot area must be readable by the user process for this to work.) This code, termed the *signal-trampoline code*, invokes the signal handler with the appropriate argument list, and, if the handler returns, makes a *sigreturn* system call to reset the process's signal state to what it was prior to the signal.

Job Control

Job control is a facility provided by the C shell [Joy, 1986] and by the Korn shell [Korn, 1983; Korn, 1984] that permits a user to control the operation of groups of processes termed *jobs*. The most important facility provided by job control is multiplexing access to the user's terminal, so that only one job at a time may read from and/or 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 windows on those systems that have them. Job control is implemented on top of the process group and signal facilities.

Figure 4.8 Delivering a signal to a process.



Each job is a collection of processes that are assigned the same process-group identifier. Processes organized in a pipeline (see Sections 2.4 and 2.6) are placed in a single process group, typically defined by the PID of the process at the head of the pipeline. Jobs are manipulated by posting of signals to the job's process group with the *killpg* system call. For example, to force a job to stop,

```
killpg(pgrp, SIGSTOP)
```

would be used.

Several jobs may compose a *session*, which is intuitively all the processes started during a single login session. A session usually has an associated *controlling terminal* that is used by default for communicating with the user. Certain characters typed at the keyboard of this terminal produce signals that are sent to all processes in one job in the session; that job is in the *foreground*, while all other jobs in the session are in the *background*. A shell may change the foreground job by using the *TIOCSPGRP ioctl* on the controlling terminal. Background jobs will be automatically stopped by a signal if they attempt to access the terminal by performing a *read*, *write*, or *ioctl* system call (with the last enforced only if the call would alter the state of the terminal).

The job-control facilities were added to the Berkeley UNIX systems in 1980. Their design was based on similar facilities included in the TENEX operating system [Bobrow *et al.*, 1972]. The original UNIX implementation was done on a PDP-11/70 by Jim Kulp at the International Institute for Applied Systems Analysis (IIASA) in Laxenburg, Austria. There is more to job control than has been described here; Chapter 9 covers the material related to terminal handling.

The job-control facilities found in AT&T System V, Release 3 attempt to provide equivalent functionality, but use a scheme based on pseudoterminals and process groups [AT&T, 1987]. POSIX has adopted job control very much like that in 4.3BSD, and the corresponding FIPS requires it. Thus, it is likely that most UNIX future systems will have job control derived from 4BSD.

4.8 Process Debugging

4.3BSD, like most UNIX systems, provides a simplistic 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 state. In particular, with *ptrace*, a parent process can perform the following operations on a child process:

- Read and write address space and registers
- Intercept signals posted to the process
- Single step and continue the execution of the process
- Terminate the execution of the process

The *ptrace* call is used almost exclusively by program debuggers, such as **adb** and **dbx**.

When a process is being traced, any signals posted to a process cause the process to enter the SSTOP state. The parent process is notified with a SIGCHLD signal and may interrogate the status of the child with the *wait* and *wait3* system calls. 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 UNIX 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 STRC bit in the *p_flag* field of the proc 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. As 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.

All the operations provided by *ptrace* are carried out in the context of the process being traced. When a parent process wants to perform an operation, it places the parameters associated with the operation into a data structure named *ipc*, clears the SWTED flag in the proc structure of the child process, and sleeps on the address of *ipc*. The next time the child process encounters a signal (immediately if it is currently stopped by a signal), it retrieves the parameters from the *ipc* structure and performs the requested operation. The child process then places a return result in the *ipc* structure and does a *wakeup()* call with the address of *ipc* as the wait channel.

The *ptrace* facility is very inefficient for three reasons. First, *ptrace* uses a single global data structure for passing information back and forth between all the parent and child processes in the system. As there is only one structure, it must be interlocked to ensure that only one parent-child process pair will use it at a time. Second, because the data structure has a small, fixed size, the parent process is limited to reading or writing 32 bits at a time. Finally, since each request by a parent process must be performed in the context of the child process, two context switches need to be done for each request—one from the parent to the child to send the request, and one from the child to the parent to return the result of the operation.

The debugging facilities included in the **/proc** filesystem, found in UNIX Eighth Edition [Killian, 1984], address all these problems. In the **/proc** system, the address space of another process can be accessed with *read* and *write* system calls, which allows a debugger to access a process being debugged with much greater efficiency. The data structure used to pass messages back and forth

between processes has been replaced by a mechanism that eliminates contention for the *ipc* structure, as well as the context switches between the parent and child processes.

Exercises

- 4.1 Why is the user structure separate from the *proc* structure?
- 4.2 What are the implications of not having the user structure mapped at a fixed virtual address in the kernel's address space?
- 4.3 Why is the performance of the context-switching mechanism critical to the performance of a highly multiprogrammed system?
- 4.4 What effect would increasing the time quantum have on the system's interactive response and overall throughput?
- 4.5 What effect would reducing the number of run queues from 32 to 16 have on the scheduling overhead and on overall system performance?
- 4.6 Give three reasons for the system to select a new process to run.
- 4.7 What type of scheduling policy does 4.3BSD use? What type of jobs does the policy favor? Propose an algorithm for identifying these favored jobs.
- 4.8 Is job control still a useful facility, now that window systems are widely available? Why or why not?
- 4.9 When and how does process scheduling interact with the memory-management facilities?
- 4.10 Define the properties of a real-time system. Give two reasons why 4.3BSD is not a real-time system.
- 4.11 After a process has exited, it may enter the state of being a zombie, SZOMB, before disappearing from the system entirely. What is the purpose of the SZOMB state? What event causes a process to exit from SZOMB?
- 4.12 Suppose the data structures shown in Fig. 4.2 did not exist. What is the minimal information that the system would need to maintain to identify all the descendents of a process? 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 descendents
 - e. destruction of a process

- 4.13 The system raises the hardware priority to *splhigh* in the *sleep()* routine before altering the contents of a process's *proc* structure. Why?
- 4.14 A process blocked with a priority less than *PZERO* may never be awakened by a signal. Describe problems this situation may cause if a disk becomes unavailable while the system is running.
- 4.15 For each state listed in Table 4.2, list the system queues on which a process in that state might be found.
- *4.16 In 4.3BSD, 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?
- *4.17 Before the 4.3BSD 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 4.3BSD. Why? How should the code be rewritten?

- *4.18 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.
- *4.19 What are the implications of adding a fixed-priority scheduling algorithm to 4.3BSD?
- *4.20 Why are signals a poor interprocess-communication facility?
- **4.21 A *kernel-stack-invalid* trap occurs when an invalid value for the kernel-mode stack pointer is detected by the hardware. Assume this trap is received on an interrupt stack in kernel mode. How might the system gracefully terminate a process that receives such a trap while executing on the kernel's run-time stack contained in the user structure?
- **4.22 Describe a synchronization scheme that would work in a tightly coupled multiprocessor hardware environment. Assume the hardware supports a *test-and-set* instruction.
- **4.23 Describe alternatives to the *test-and-set* instruction that would allow you to build a synchronization mechanism for a multiprocessor UNIX system.

- **4.24 A *lightweight process* is a thread of execution that operates within the context of a normal UNIX process. Multiple lightweight processes may exist in a single UNIX 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.
- **4.25 Describe new kernel facilities that would be useful in implementing lightweight processes. How does the existing UNIX process context organization make kernel support for lightweight processes difficult?

References

AT&T, 1987.

AT&T, *The System V Interface Definition (SVID)*, Issue 2, American Telephone and Telegraph, Murray Hill, NJ (January 1987).

Bobrow *et al.*, 1972.

D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, & R. S. Tomlinson, "TENEX, a Paged Time Sharing System for the PDP-10," *Comm ACM* **15**(3), pp. 135–143, Association for Computing Machinery (March 1972).

Ferrin & Langridge, 1980.

T. E. Ferrin & R. Langridge, "Interactive Computer Graphics with the UNIX Time-Sharing System," *Computer Graphics* **13**, pp. 320–331 (1980).

Joy, 1986.

W. N. Joy, "An Introduction to the C Shell," pp. 4:1–46 in *UNIX User's Supplementary Documents, 4.3 Berkeley Software Distribution, Virtual VAX-11 Version*, USENIX Association, Berkeley, CA (1986).

Killian, 1984.

T. J. Killian, "Processes as Files," *USENIX Association Conference Proceedings*, pp. 203–207 (June 1984).

Korn, 1983.

D. Korn, "KSH—A Shell Programming Language," *USENIX Association Conference Proceedings*, pp. 191–202 (June 1983).

Korn, 1984.

D. Korn, *Introduction to the Korn Shell*, Issue 2, AT&T Bell Laboratories, Murray Hill, NJ (November 1984).

Ritchie, 1988.

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

Sanderson *et al.*, 1986.

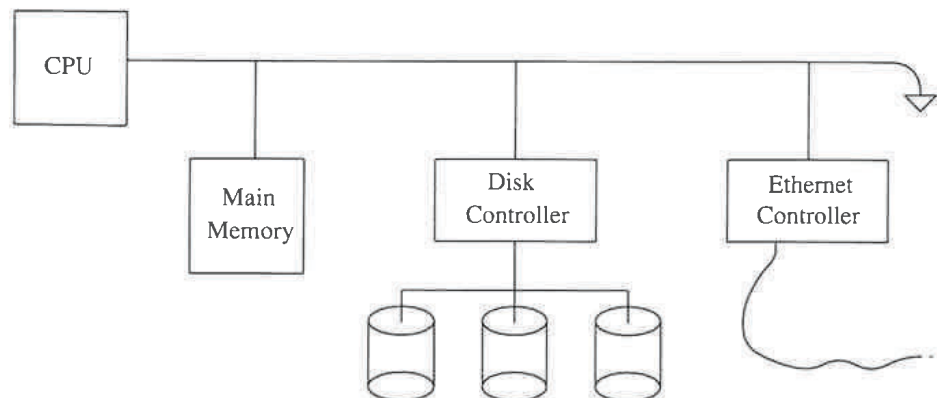
T. Sanderson, S. Ho, N. Heijden, E. Jabs, & J. L. Green, "Near-Realtime Data Transmission During the ICE-Comet Giacobini-Zinner Encounter," *ESA Bulletin* **45**(21) (1986).

Memory Management

5.1 Terminology

A central component of any operating system is the *memory-management system*. As the name implies, memory-management facilities are responsible for the management of memory resources available on a machine. These resources are typically layered in a hierarchical fashion, with memory access times inversely related to their proximity to the CPU, as shown in Fig. 5.1. The primary memory system is termed *main memory*; the next level of storage is referred to as *secondary storage* or *backing storage*. Main-memory systems usually are constructed from random-access memories, whereas secondary stores are placed on moving-head disk drives. In some workstation environments, the common two-level hierarchy is

Figure 5.1 Hierarchical layering of memory.



becoming a three-level hierarchy, with the addition of file-server machines connected to a workstation via a local-area network [Gingell *et al.*, 1987].

In a multiprogrammed environment, it is critical for the operating system to share available memory resources effectively among the processes. The operation of any memory-management policy is directly related to the memory required for a process to execute. That is, if a process must reside entirely in main memory in order for it to execute, then a memory-management system must be oriented toward allocating large units of memory. On the other hand, if a process can execute when it is only partially resident in main memory, then memory-management policies are likely to be very different. Memory-management facilities usually try to optimize the number of runnable processes that are resident in main memory. This goal must be considered together with the goals of the process scheduler (Chapter 4), in order to avoid conflicts that can adversely affect overall system performance.

Although the availability of secondary storage permits more processes to exist than can be resident in main memory, it also requires additional algorithms that can be complicated. Space management typically requires algorithms and policies different from those used for main memory, and a policy must be devised for deciding when to move processes between main memory and secondary storage.

Processes and Memory

Each process operates on a *virtual machine* that is defined by the architecture of the underlying hardware on which it executes. We are interested in only those machines that include the notion of a *virtual address space*. A virtual address space is a range of memory locations that a process references independently of the physical memory present on the hardware. References to the virtual address space—*virtual addresses*—are translated by hardware into references to physical memory. This operation, termed *address translation*, permits programs to be loaded into memory at any location without requiring position-dependent addresses in the program to be changed. Address translation and virtual addressing are also important in efficiently sharing a CPU, because position-independence usually permits context switching to be done very quickly.

Most machines provide a contiguous virtual address space for processes. Some machines, however, choose to partition visibly a process's virtual address space into regions termed *segments* [Intel, 1984]; such segments usually must be physically contiguous in main memory and begin at fixed addresses. We shall be concerned with only those systems that do not visibly segment their virtual address space. This use of the word *segment* is not the same as its earlier use when we were describing UNIX process segments, such as text and data segments; the latter segments are variable in size.

When multiple processes are coresident in main memory, the physical memory associated with each process's virtual address space must be protected to ensure that one process cannot alter the contents of another process's virtual address space. This protection is implemented in hardware and is usually tightly

coupled with the implementation of address translation. Consequently, the two operations usually are defined and implemented together as hardware termed the *memory-management unit*.

When the effective range of addressable memory locations provided to a process is independent of the size of main memory, we say the machine supports *virtual memory*. Another way of saying this is that the virtual address space of a process is independent of the physical address space of the CPU. For a machine to support virtual memory, we also require that the whole of a process's virtual address space need not be resident in main memory for that process to execute.

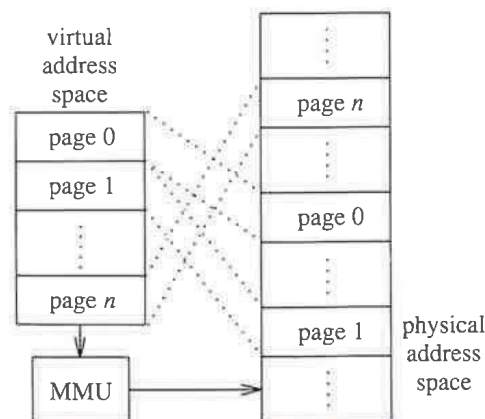
Virtual memory can be implemented in many ways, some of which are software-based, such as *overlays*. Most effective virtual-memory schemes are, however, hardware-based. In these schemes, the virtual address space is divided into fixed-sized units, termed *pages*, as shown in Fig. 5.2. Virtual-memory references are resolved by the address-translation unit to a page in main memory and an offset within that page. Hardware protection is applied by the memory-management unit on a page-by-page basis.

Some systems provide a two-tiered virtual memory system in which pages are grouped into segments [Organick, 1975]. In these systems, protection is usually at the segment level. In the remainder of this chapter, we shall be concerned with only those virtual-memory systems that are page-based.

Paging

Address translation handles the first requirement of virtual memory by decoupling the virtual address space of a process from the physical address space of the CPU. To satisfy the second requirement, each page of virtual memory is marked as *resident* or *nonresident* in main memory. If a process references a location in virtual memory that is not resident, a hardware trap termed a *page fault* is generated.

Figure 5.2 Paged virtual-memory scheme. MMU—memory-management unit.



The servicing of page faults, or *paging*, permits processes to execute even if they are only partially resident in main memory.

Coffman and Denning [1973] characterize paging systems by three important policies:

1. When the system loads pages into memory (the *fetch policy*)
2. Where the system places pages in memory (the *placement policy*)
3. How the system selects pages to be removed from main memory when pages are unavailable for a placement request (the *replacement policy*)

In normal circumstances, all pages of main memory are equally good and the placement policy has no effect on the performance of a paging system. Thus, a paging system's behavior is dependent on only the fetch policy and the replacement policy. Under a *pure demand-paging* system, a demand-fetch policy is used, in which only the missing page is fetched and replacements occur only when main memory is full. Consequently, the performance of a pure demand-paging system depends on only the system's replacement policy. In practice, paging systems do not implement a pure demand-paging algorithm. Instead, the fetch policy often is altered to do *prepaging*—the fetching of pages of memory other than the one that caused the page fault—and the replacement policy is invoked before main memory is full.

Replacement Algorithms

The replacement policy is the most critical aspect of any paging system. There is a wide range of algorithms from which we can select in designing a replacement strategy for a paging system. Much research has been carried out in evaluating the performance of different page-replacement algorithms [Belady, 1966; King, 1971; Marshall, 1979]. In preparation for our discussion of the replacement algorithm used in 4.3BSD, we shall introduce some basic terminology.

A process's paging behavior for a given input is described in terms of the pages referenced over the time of the process's execution. This sequence of pages, termed a *reference string*, represents the behavior of the process at discrete times during the process's lifetime. Corresponding to the sampled references that constitute a process's reference string are real-time values that reflect whether or not the associated references resulted in a page fault. A useful measure of a process's behavior is the *fault rate*, which is the number of page faults encountered while processing a reference string, normalized by the length of the reference string.

Page-replacement algorithms typically are evaluated in terms of their effectiveness on reference strings that have been collected from executing real programs. Formal analysis can also be used, although it is difficult to perform unless many restrictions are applied to the execution environment. The most common metric used in measuring the effectiveness of a page-replacement algorithm is the fault rate.

Page-replacement algorithms are defined in terms of the criteria they use for selecting pages to be reclaimed. For example, the *Optimal Replacement Policy* [Denning, 1970] states that the “best” choice of a page to replace is the one with the longest expected time until its next reference. Clearly, this policy is not applicable to dynamic systems, as it requires a priori knowledge of the paging characteristics of a process. The policy is useful for evaluation purposes, however, as it provides a yardstick for comparing the performance of other page-replacement algorithms.

Practical page-replacement algorithms require a certain amount of state information that is used in selecting replacement pages. This state typically includes the reference pattern of a process, sampled at discrete time intervals. On some systems, this information can be expensive to collect [Babaoğlu & Joy, 1981]. As a result, the “best” page-replacement algorithm may not be the most efficient.

Working-Set Model

The working-set model assumes that processes exhibit a slowly changing locality of reference. For a period of time, a process operates in a set of subroutines or loops, causing all its memory references to refer to a fixed subset of its address space, termed the *working set*. The process periodically changes its working set, abandoning certain areas of memory and beginning to access new ones. After a period of transition, the process defines a new set of pages as its working set. In general, if the system can provide the process with enough pages to hold the pages in that process’s working set, the process will experience a low page-fault rate. If the system cannot provide the process with enough pages for the working set, the process will run slowly and will have a high page-fault rate.

Precisely calculating the working set of a process is impossible without a priori knowledge of that process’s memory reference pattern. However, the working set can be approximated by various means. One method of approximation is to track the number of pages held by a process and that process’s page-fault rate. If the page-fault rate increases above a high watermark, the working set is assumed to have increased and the number of pages held by the process is allowed to grow. Conversely, if the page-fault rate drops below a low watermark, the working set is assumed to have decreased and the number of pages held by the process is contracted.

Swapping

Swapping is the term used to describe a memory-management policy in which entire processes are moved to and from secondary storage when main memory is in short supply. Swap-based memory-management systems usually are less complicated than are demand-paged systems, since there is less bookkeeping to do. However, pure swapping systems are typically less effective than are paging systems, since the degree of multiprogramming is lowered by the requirement that processes be fully resident to execute. Swapping is sometimes combined with paging in a two-tiered scheme, whereby paging is used to satisfy memory

demands until a severe memory shortfall requires drastic action, in which case swapping is used.

Secondary Storage

In this chapter, a portion of secondary storage that is used for paging and/or swapping is termed a *swap area* or *swap space*. The hardware devices on which these areas reside are termed *swap devices*.

Advantages of Virtual Memory

There are several advantages to the use of virtual memory on computers capable of supporting this facility properly. Virtual memory allows large programs to be run on machines with main memory configurations that are smaller than the program size. On machines with a moderate amount of memory, it allows more programs to be resident in main memory to compete for CPU time, as the programs need not be completely resident. When programs use sections of their program or data space for some time, leaving other sections unused, the unused sections need not be present. Also, the use of virtual memory allows programs to start up faster, as they generally require only a small section to be loaded before they begin processing arguments and determining what actions to take. Other parts of a program may not be needed at all during individual runs. As a program runs, additional sections of its program and data spaces are paged in on demand (*demand paging*). Finally, there are many algorithms that are more easily programmed by sparse use of a large address space than by careful packing of data structures into a small area. Such techniques are too expensive for use without virtual memory, but may run much faster when that facility is available, without using an inordinate amount of real memory.

On the other hand, the use of virtual memory can degrade performance. It is more efficient to load a program all at one time than to load it entirely in small sections on demand. There is a finite cost for each operation, including saving and restoring state and determining which page must be loaded. So, some systems use demand paging only for programs larger than some minimum size.

Hardware Requirements for Virtual Memory

Nearly all versions of UNIX have required some form of memory-management hardware to support transparent multiprogramming. To protect processes from modification by other processes, the memory-management hardware must prevent programs from changing their own address mapping. The UNIX kernel runs in a privileged mode (*kernel mode* or *system mode*) in which memory mapping may be controlled, whereas processes run in an unprivileged mode (*user mode*). There are several additional architectural requirements for support of virtual memory. The first and most obvious is that the CPU must distinguish between resident and nonresident portions of the address space, must suspend programs when they refer to nonresident addresses, and must resume their operation once the operating system has placed the required section in memory. As the CPU may discover missing

data at various times during the execution of an instruction, it must provide a mechanism to save the machine state so that the instruction can be continued or restarted later. The CPU may implement restarting by saving enough state when an instruction begins that the state can be restored when a fault is discovered. Alternatively, instructions could delay any modifications or side effects until after any faults would be discovered, so that the instruction execution need not back up before restarting. On some computers, instruction backup requires the assistance of the operating system.

Most machines designed to support demand-paged virtual memory include hardware support for the collection of information on program references to memory. When the system selects a page for replacement, it must save the contents of that page if they have been modified since the page was brought into memory. The hardware usually maintains a per-page flag showing whether the page has been modified. Many machines also include a flag recording any access to a page for use by the replacement algorithm.

5.2 Evolution of 4.3BSD Memory Management

The 4.3BSD memory-management system has evolved from early releases of UNIX on the PDP-11 minicomputer. Before we present the 4.3BSD memory-management system, we shall review aspects of some of its predecessors.

Version 7 UNIX

Version 7 was the last version of UNIX that was released from Bell Laboratories for the PDP-11, a 16-bit minicomputer. The PDP-11 uses a segmented memory architecture. The virtual-address space of a process is divided into eight segments that can be up to 8 Kbyte each. This virtual-address-space limit of 64 Kbyte (2^{16} bytes) applies not only to each user process but also to the kernel (which executes in a separate, privileged protection mode).

Early PDP-11s supported a maximum of 248 Kbyte of main memory; later models, such as the PDP-11/70, allowed nearly 4 Mbyte of memory. Some models also supported separate instruction and data spaces, a feature that effectively doubled the maximum program size.

The limited virtual-address range of the PDP-11 inspired developers at Case Western Reserve, Purdue University, the University of California at Berkeley, and several European universities to create software-overlay schemes for both user programs and the kernel. Overlays for the kernel were especially useful because the limited address space was usually the biggest problem with providing facilities such as networking on the PDP-11. Even with software overlays, however, the maximum size of individual programs on the PDP-11 was small in comparison to the amount of main memory that was usually available. And overlays extended only the effective text space, not the data space (at least, not without losing addressing transparency). Larger data space is what many programs, such as those written in LISP, really need.

UNIX memory-management techniques were particularly simple, as befitted this class of machine. Programs were loaded contiguously in physical memory and in their entirety. Memory sharing was limited to read-only sharing of text segments in suitably marked programs. Memory management was swap-based, with the secondary storage used for swapping limited to a single dedicated area of a disk. Finally, swap space for processes was allocated at the time a process was created, to ensure its availability.

UNIX 32V

The VAX-11/780 was introduced in 1978. This 32-bit minicomputer has a 4-gigabyte (2^{32} -byte) virtual-address space and was designed to support demand-paged virtual memory.

The first UNIX system that ran on a VAX was called 32V; it was a straightforward port of the existing PDP-11 version of UNIX. The developers created this port initially by emulating PDP-11 hardware memory-management facilities on the VAX's paged virtual-memory architecture [London & Reiser, 1978]. Processes were still fully resident and contiguous in main memory. The only restriction that was eased was the maximum size of a program; this limit was increased from 64 Kbyte to a tunable limit that was, by default, 192 Kbyte. Although programs could be larger than on the PDP-11, they still had to be fully resident. Therefore, programs were limited in size by the amount of physical memory present on a machine (0.5 to 2 Mbyte, although part of that memory was occupied by the UNIX kernel, which was permanently resident).

The original port also followed Version 7 in that entire processes were still swapped to secondary storage when main memory was in short supply. In addition, the only memory sharing among processes was accomplished through pure text segments.

Subsequent revisions to the memory-management system before release of 32V expanded on this initial scheme in two ways. First, contiguous layout of programs in main memory was replaced by a *scatter-loading* scheme whereby a process's virtual-address space was loaded into main memory on a page-by-page basis. Second, when main memory was needed, parts of a process, instead of entire processes, could be swapped to secondary storage. Although these two facilities avoided memory fragmentation and provided a higher degree of multiprogramming than was available on the PDP-11, respectively, they did not fully support the VAX's virtual-memory architecture. It still was not possible to create a process whose virtual memory was larger than the physical memory available on a machine, because the entire process was required to be resident in order for that process to execute.

3BSD

The 3BSD release replaced the scatter loading and partial swapping found in 32V with a demand-paged virtual-memory-management system. The design of this system was motivated by a need to run very large processes (larger than the

available physical memory), and to provide an accessible environment for carrying out performance evaluation studies related to virtual-memory management [Babaoğlu *et al.*, 1979].

The 3BSD paging system used

- A demand-paging fetch policy (i.e., only those pages that were faulted were fetched by the system)
- A page-by-page placement policy
- A replacement algorithm that approximated global least recently used (LRU)

The replacement algorithm considered all available physical memory as a loop of pages, scanning the loop to collect reference information, and selecting replacements according to least recent use. In addition, memory-utilization statistics maintained by the paging system were used by the process scheduler to vary dynamically the degree of multiprogramming.

As in previous systems, all of the kernel text and most of the kernel data resided permanently in main memory. The structure of the virtual-address space of a process (text, data, and stack segments) remained unchanged, although the space's maximum size was no longer limited by the available memory. Memory sharing among processes also was unchanged, due mainly to an underlying design decision to minimize user-visible changes to the system.

Secondary-storage management was not significantly different in 3BSD from that in 4.3BSD. Storage was preallocated for the entire address space of a process at the time the process was created. This policy was mainly a result of the large amount of secondary storage in comparison to main memory.

4.1BSD

The memory-management system in the 4.0BSD and 4.1BSD releases was the result of concentrated performance-tuning work [Kashtan, 1980; Joy, 1980]. The fundamental algorithms and data structures introduced in 3BSD remained virtually unchanged. Alterations were made, however, to address specific performance problems and to track hardware changes, such as the introduction of the VAX-11/750.

4.1BSD contained five changes of note:

1. The paging system was altered such that, instead of manipulating hardware pages, it dealt with a logical page, termed a *cluster*. Clusters were required to be multiples of the underlying hardware page size and were added to reduce the cost of page-related operations, such as page-fault handling. The use of clustering also reduced the size of certain system data structures, thereby permitting larger virtual memories to be supported in the same amount of physical memory.

2. A limited amount of prepaging was added to the fetch policy. Part of the prepaging was a consequence of the addition of clustering. Instead of fetching a single page in response to a fault, the system fetched as many pages as were in the cluster.
3. The overhead associated with moving dirty pages to secondary storage was reduced by multiple clusters being written in a single operation whenever possible.
4. The device and block numbers of pages from text images were stored with the descriptions of free pages, caching text pages that had been used recently in memory.
5. Processes were provided a minimal amount of control over the page-replacement and prepaging policies used by the system. This control took the form of a system call that a process could use to notify the system when it was expecting to exhibit sequential or random memory reference patterns. The system then used this information to disable prepaging (in the case of random accesses), or to mark pages as likely candidates for replacement.

4.3BSD

The changes to the virtual-memory system that appeared in 4.3BSD were oriented toward adapting to changes in hardware technology.

- The text images and page tables of all programs were retained in a cache after *exit*, much as in the older *sticky-bit* mechanism that was used to force pages of heavily used programs to be retained in swap space.
- The page-replacement algorithm was modified to be independent of the size of main memory; this change improved the performance of the paging system on machines with large main memories.
- Workstations that utilized remote file servers, combined with larger main memories, made it worthwhile to expand the amount of prepaging.

5.3 VAX Memory-Management Hardware

This section provides an overview of the VAX memory-management architecture, on which the design of the 4.3BSD virtual-memory system is heavily dependent. Further information on the VAX hardware can be found in the *VAX Hardware Handbook* [DEC, 1980]. Memory management on the Tahoe processor is similar to that on the VAX.

VAX Virtual Address Space

The VAX memory-management hardware supports a two-level mapping mechanism for doing address translation. The first-level page tables reside in the

system's virtual address space and map user page tables. These tables in turn, map the user virtual address space. The 32-bit virtual address space of the VAX is divided into four equal-sized blocks, as shown in Fig. 5.3.

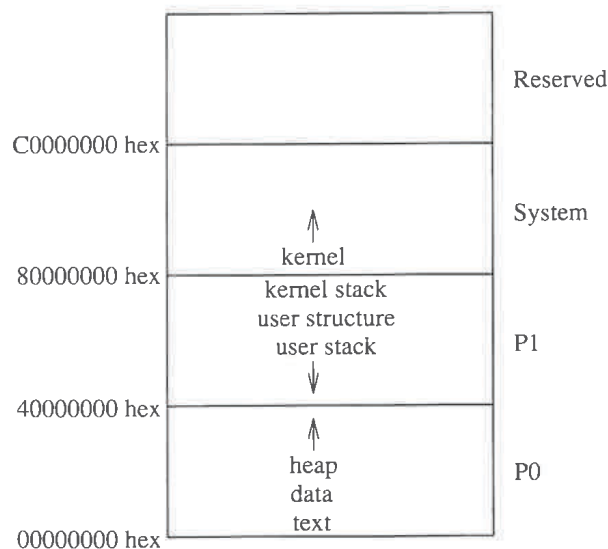
Two of these blocks, known as the *P0* and *P1* regions, are used for the address space of a user process. The third block, known as the *system region*, contains the shared kernel virtual address space; the fourth region (marked *reserved*) is not supported by current VAX hardware. The P0 region starts at virtual address 0 and grows toward higher addresses. The P1 region on the other hand, starts at the top of the user virtual address space and grows toward lower addresses. The regions are described by two per-process base-length register pairs. The address space on the Tahoe uses all four regions, with the system in the top region.

On the VAX, the virtual address space of the current user process is directly addressable by the kernel. In general, the system takes advantage of this accessibility in only low-level parts of the kernel—for example, in the *copyin()* and *copyout()* routines that move data between user and kernel space. Use in such clearly defined routines provides efficiency benefits, while avoiding more widespread use that would inhibit portability. The efficiency advantages are large enough that some non-VAX implementations, such as Sun Microsystems MC68000 machines, also map user space into the kernel.

VAX Page Tables

The VAX page size is fixed at 512 bytes. Each VAX *page-table entry* (PTE) consists of four bytes of mapping and protection information, as shown in Fig. 5.4. If the hardware attempts to translate an address through a PTE that has the *valid* bit

Figure 5.3 Layout of VAX virtual address space.



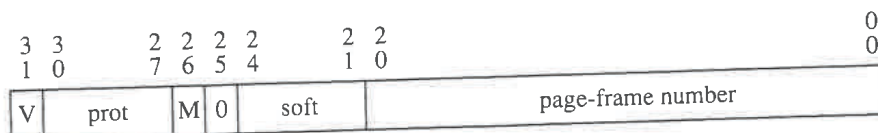


Figure 5.4 Format of a VAX page-table entry (PTE). V—page valid bit; prot—protection-mode field; M—page-modified bit; soft—software defined field.

off, a *translation-not-valid fault* (i.e., a *page fault*) is generated. Whereas most architectures that support virtual memory, including the Tahoe, provide a per-page *reference bit* that is automatically set by the hardware when the corresponding page is referenced, the VAX has no such mechanism. The mechanism used by 4.3BSD to simulate a reference bit is discussed in Section 5.12.

The fields in a PTE contain the physical page address that holds the data for that address, as well as access modes that specify whether a page may be read and/or written. A four-bit portion of each PTE is reserved for use by software; it is not interpreted by the VAX memory-management hardware. Finally, one bit of the PTE, the *modify* bit, is set by the hardware each time the page is modified.

VAX page tables reside in main memory when they are being used. The hardware supports paging of page tables, but this facility is not utilized by 4.3BSD.

System-Address Translation

Each region of the address space is mapped by a separate page table. System space is mapped by the system page table, which is physically contiguous in main memory. Two hardware registers are set to the address and length of the system page table once it has been set up at boot time. The *system base register* (SBR) contains the physical address of the *system page table*, and the *system length register* (SLR) contains the number of entries in the page table. In 4.3BSD, the system page table is statically allocated, and the size of the system map is thus set when the kernel is compiled.

Since the 32-bit VAX address space is divided into four equal-sized regions, the two most significant bits of the address determine the region into which an address falls (see Table 5.1).

Once a region has been selected, the remaining address bits are interpreted as a page number within the region and a byte offset within the page. Since the pages are 512 bytes each, the low-order nine bits give the byte offset within the page. The remaining 21 bits specify the page index in the region.

Virtual-address translation for kernel addresses is shown in Fig. 5.5. Once an address has been identified as a part of system space according to bits 30 and 31 of the address, the page index is extracted from bits 9 through 29. The index is checked against the system length register, and, if valid, is added to the system base register to find the PTE. As the memory-management hardware translates the address, it locates the PTE, checks that the page is resident and that the desired access is allowed, and then extracts the physical page number. Finally, the

Table 5.1 Mapping of high-order bits to regions.

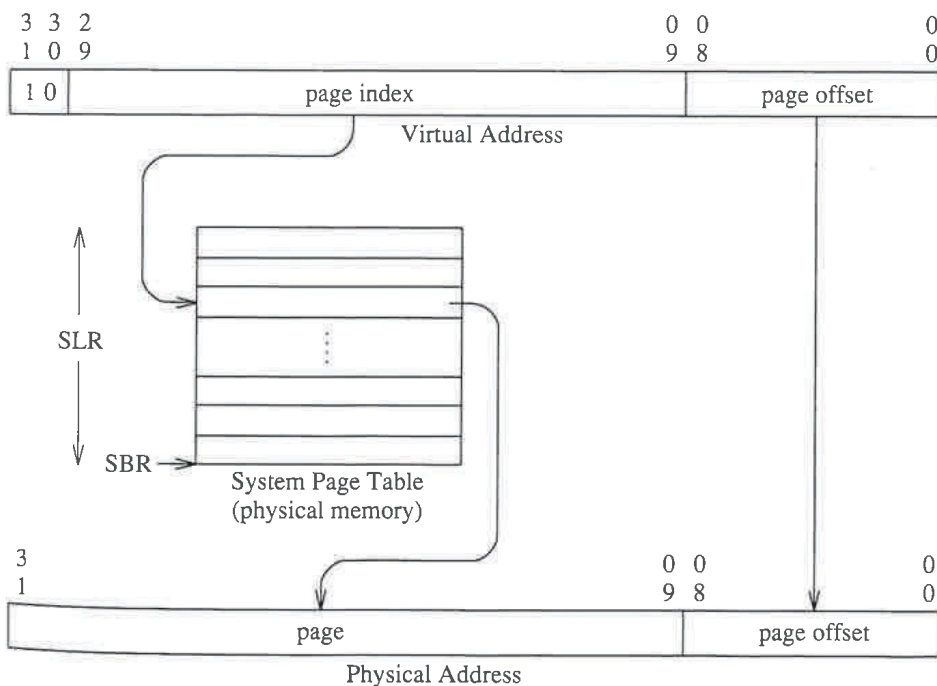
High-order address bits	Region
11	reserved
10	system
01	P1
00	P0

physical page number is combined with the byte offset within the page to generate a physical-memory address.

User-Address Translation

Unlike the system page tables, the user page tables (mapping the P0 and P1 regions) are located in virtual rather than physical memory. Because a four-byte entry is needed to map each 512-byte page of address space, a map for one of these 1-gigabyte regions would require 8 Mbyte of page tables if all the entries

Figure 5.5 System virtual-address translation. SLR—system length register; SBR—system base register.



mapping the region were created. However, the hardware locates these page tables in kernel virtual memory, and therefore not all pages of the page table need to be resident simultaneously (although the system does not take advantage of this capability). As a result, the virtual-address-translation process for user addresses is a two-step translation using a two-level page table.

Like the system page table, the P0 page table is described by two hardware registers. The *P0 base register (POBR)* is the system virtual address of the start of the page table for the P0 region for the current process. The *P0 length register (POLR)* contains the number of entries in the P0 table. These registers are set from the PCB on each context switch, as they are generally different for each process. The P1 region is described in a similar way by the *P1 base register (PIBR)*, and by the *P1 length register (PILR)*. Both base registers point to the beginning of the map for a 1-gigabyte region. However, whereas the POLR specifies the number of PTEs present in the P0 map, the P1 length register specifies the number of entries that are *outside* the region. This scheme is optimized for segment growth, since expansion requires only a change in the length register.

The procedure for virtual-address translation for a reference to a P0 or P1 address is similar to the procedure for system-address translation, except for the location of the page tables. The high-order bits of the address select a region. Then, as with a system address, the page index is extracted from bits 9 through 29 of the address. This index is compared with the appropriate length register. If the index is less than the value of the length register for P0 space (greater than or equal to the length register for P1 space), the page value is valid. The index is then combined with the base register to yield a system virtual address for the PTE. Figure 5.6 shows the translation that indexes into the table to find the entry for the desired page. This system address is translated as described previously to calculate a physical address from which to fetch the user PTE. Finally, the PTE is checked for validity and protection of the page, and to obtain the physical page number. The page offset is added to the page address to produce the final physical address.

Note that the location of user (P0 and P1) page tables in kernel virtual memory means that two translations are needed to find a user physical address: one translation using the system page table and one using the P0 or P1 page table. This translation mechanism can be regarded as a two-level page table for user addresses. Because the user page tables are allocated in pages and are mapped by the system page table, the page index (bits 9 through 29 of the user virtual address) can be considered to have two parts. Bits 16 through 29 select a page of PTEs, and bits 9 through 15 select a PTE in the page. Thus, one kernel PTE maps one 512-byte page of kernel memory containing 128 user PTEs, which, in turn, map 64 Kbyte of user memory. System-address translation uses only one page table, however, so the system page table has only a single level of translation.

Page Faults

While translating a virtual address, the memory-management hardware may discover a PTE that does not have the valid bit set. When such a PTE is found, the

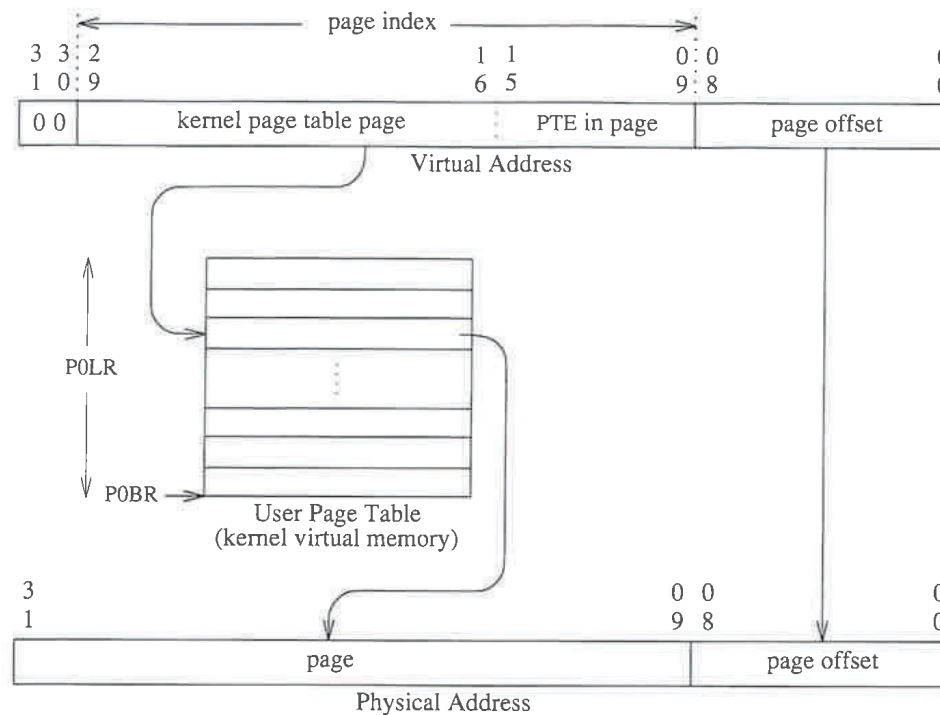


Figure 5.6 User virtual-address translation. POLR—process length register; POBR—process base register.

current instruction cannot be completed until the needed page is brought into memory and the PTE is marked valid. The memory-management hardware initiates a *page fault* by executing a trap sequence to system mode. The pending instruction is backed up to its beginning, so that it can be restarted. Restarting requires that any side effects, such as register or memory modifications, be undone so that they will be performed only once. (A few complex instructions, such as string moves, may set the state, so that the instruction can be resumed rather than restarting.) When a fault occurs, the operating system gains control and analyzes the fault. In the normal case, a page is fetched from disk or a new page is allocated for the process. After updating the PTE, the system returns from the trap. The return to user mode allows the CPU to reattempt execution of the original instruction.

The memory-management hardware may discover other conditions that prevent an address translation from completing successfully. A page index may be outside the range specified by the appropriate length register, or an access may not be allowed by the protection field of a PTE. If the hardware detects either of these conditions, it causes a trap to the kernel instead of completing the address translation. Most such traps cause the kernel to deliver a signal (SIGSEGV or SIGBUS) to the process rather than to continue execution. There is one exception:

When the stack pointer is below the current stack segment but still within the P1 region, the kernel automatically extends the stack and returns to the user program.

Translation Buffers

The use of a two-level page table and the table's location in main memory imply that as many as three memory references may be required to fetch the contents of a virtual address. As this process would be prohibitive if it were required on each memory reference, a hardware cache of PTEs, termed a *translation lookaside buffer*, is maintained. When an address translation is performed, the translation buffer is checked for a matching entry, thus avoiding memory accesses. If the required translation is not already in the cache, it is fetched and is placed in the cache, possibly displacing an older entry.

The presence of the translation cache requires that the operating system purge any *stale translations* when PTEs are modified in memory. Purges may be done on the VAX through invalidation of the entire cache by moving a zero to the *translation buffer invalidate all* (TBIA) register, or through invalidation of an entry for a single virtual address by moving the address to the *translation buffer invalidate single* (TBIS) register.

5.4 Management of Main Memory: The Core Map

In 4.3BSD, main memory is logically divided into two parts. The kernel is permanently resident in memory and uses memory to hold its text segment and most of its data segment. In addition, certain kernel data structures, such as the process table, that also must remain resident are allocated at the time the kernel is initialized. The remainder of main memory is allocated to processes according to demand (see Fig. 5.7). (A small area at the top of memory is used to hold the last few messages that were printed on the console; this arrangement allows postmortem debugging after a system crash.)

The *core map* is the central data structure used to manage main memory. It consists of an array of structures, one entry for each cluster of pages in main memory, excluding those allocated to the kernel text and data. The core map is allocated at boot time as one of the last steps in the bootstrap procedure. The core map manages a contiguous section of main memory from *firstfree* at the end of the kernel data segment to *maxfree*, as shown in Fig. 5.7. This contiguous region is the part of main memory available to user processes.

Figure 5.7 Physical memory available to processes. Msgs—messages.



The core map is maintained as a contiguous array of structures, one per page cluster. Although the elements are small (16 bytes in 4.3BSD), if there are many of them, the overall size may be large. Thus, it is desirable for the page-cluster size to be large to reduce the number of elements and the size of the array. The definition of the core-map structure is given in Fig. 5.8.

Because there is a one-to-one correspondence between the core-map array and the page clusters of main memory available for allocation, it is simple to locate a page cluster given a core-map structure, and vice versa. Thus, it is common for the memory-management routines in 4.3BSD to use core-map structures as a means of referencing portions of main memory.

A core-map structure is used for several different functions:

- The structure records information about the current use of a page cluster. This information can be used to map physical-memory addresses to virtual-memory addresses.
- Core-map elements for free pages are linked to form the memory free list.
- Locking and synchronization on a per-cluster basis is done through core-map structures.

Figure 5.8 Definition of the core-map structure.

```
struct cmap {
    unsigned short c_next,      /* index of next free-list
                                entry */
                c_prev,      /* index of previous
                                free-list entry */
                c_hlink;     /* hash link for <blkno,mdev> */
    unsigned short c_ndx;      /* index of owner proc or text */
    unsigned int  c_page:21,   /* virtual page number
                                in segment */
                c_lock:1,    /* locked for raw i/o or pagein */
                c_want:1,    /* wanted */
                c_intrans:1, /* intransit bit */
                c_free:1,    /* on the free list */
                c_gone:1,    /* associated page has
                                been released */
                c_type:2,    /* type CSYS or CTEXT or CSTACK
                                or CDATA */
                :4,         /* to longword boundary */
                c_blkno:24,  /* disk block this is a copy of */
                c_mdev:8;    /* which mounted dev
                                this is from */
};
```

- Free pages with useful contents are located using the core map.

The next few sections elaborate on each of these functions.

Physical-to-Virtual Translation

The first function of the core map is to store information about the current usage of the corresponding pages. This information is stored in a compact form to minimize space. The type of a page in its core-map entry is any one of system, text, data, or stack. Core-map entries for system pages need not contain other information, as they are immune to the operations done on the core map. Otherwise, the entry contains an index into the text table (for text pages) or into the process table (for data and stack pages and executables with writeable text segments) to identify the current user of a page cluster. The core-map entry also contains a page index describing the position of the page in the virtual segment that contains that page. These two items are sufficient to locate the PTEs for the process(es) using the memory associated with a core-map structure. Thus, information in the core-map structure permits translation of physical address to virtual address, much as the page tables support translation in the other direction.

Memory Free List

The second use of the core map is in creating the *memory free list*. All unallocated clusters of main memory are chained together using a doubly linked list formed from two fields of the core-map structure. To save space, the kernel constructs these chains using 16-bit indices, instead of 32-bit pointers (at least on the VAX).

The free list is kept in least-recent-use (LRU) order, and acts as a cache of potentially useful data. That is, clusters with useful contents may be reclaimed from the free list until they are overwritten. Because of its use as a cache, the free list is maintained as a doubly linked list. A flag in each core-map structure, *c_free*, indicates whether the cluster is on the free list.

Synchronization

The core map serves as the point of synchronization for virtual-memory functions that operate on individual page clusters. Hence, two synchronization flags (*c_lock* and *c_want*) are stored there. Memory pages are locked during the service of a page fault while I/O is being done; they are also locked during raw I/O operations that read or write directly to or from the user-process pages. The page-replacement algorithm (yet to be described) must write *dirty pages* to the swap area before freeing them; the page-replacement daemon locks such pages during the operation to avoid confusion with the page-fault handler, *pagein()*.

An additional flag is used during page-fill operations for synchronization among processes sharing a page. This flag, the *in-transit flag*, (*c_intrans*), is set between the time that a read is initiated on a page cluster and the time that the cluster is ready for use. If multiple users of a nonresident cluster attempt to use that cluster at nearly the same time, this flag prevents any process but the first from starting the read.

Text-Page Cache

As mentioned in the previous section, free pages may hold useful data. One particular case—pages of a pure text segment—requires special-purpose information to be maintained. This information is used to identify the file from which pages were originally obtained. File information is recorded in a core-map structure as a <device, block number> pair. The device information specifies either a mounted filesystem, using the associated index in the filesystem mount table, or a pseudo-mounted-device index reserved for the swap device. The block number is the filesystem block number from which the data were obtained.

Core-Map Limits

The dense packing of the core-map structure forces many fields to be limited in size. As a result, the core-map structure is the major limitation on the sizes of other objects that may be supported by the system. The core map places limits on

- The amount of main memory that can be supported (through the link fields)
- The number of mounted filesystems (through the filesystem index used for identifying the origin of an associated page cluster)
- The size of filesystems, including interleaved swap space (through the filesystem block-number field)
- The number of processes and shared text images (through the field used to index in the process and shared text tables)

Memory-Allocation Routines

Main memory is allocated through one of two routines, *memall()* and *vmemall()*. These routines differ in only one respect: *memall()* will return failure if there is not enough main memory available, whereas *vmemall()* will wait for memory to be freed, so that it always succeeds; the latter uses *memall()* to do the actual allocation. As newly allocated memory must always be mapped into some virtual address before it can be used, these routines operate by filling in a virtual-address range. The *memall()* routine is called with a pointer to a PTE mapping the start of the range, a page count, and a process and type for use in setting up core-map entries. For each page cluster in the range, *memall()* finds memory by removing the first core-map entry from the memory free list. If a previous user of the page still has a reference to the page, the corresponding PTEs are located and cleared. The page numbers are then recorded in each PTE in the cluster, and the core-map entry is initialized with the information supplied by the caller.

Two interfaces are provided for freeing memory. The *memfree()* routine reclaims all memory assigned to a range of virtual addresses, presuming all the pages are resident. An alternate interface, provided by *vmemfree()*, reclaims only that memory that is associated with resident pages in a range of virtual addresses; it ignores nonresident pages.

The main-memory pool is initialized at boot time to include all pages not used statically by the kernel. Pages for user processes and for page tables are allocated from this pool. Certain parts of the kernel, such as the network, also allocate memory from this pool. However, because the network often needs to allocate memory at interrupt level, it is important that memory be available to avoid losing packets. For this reason, and for reasons that are described in Section 5.12, the kernel attempts to maintain at least a minimal amount of free memory (the amount is a configurable parameter). The current amount of free memory is kept in a global variable, *freemem*, that can be checked by routines that need memory and cannot block. This variable is also used for synchronization: Processes may sleep on the address of *freemem* awaiting more memory; they will be awakened when memory is freed after a shortage.

5.5 Management of Swap Space

Early versions of UNIX reserved a single area of one disk as a swap area. Processes were swapped to and from this area if all active processes could not fit into main memory. In general, swap activity was low; systems were configured to avoid swapping, because system performance was unacceptable when active processes had to be swapped. The addition of virtual memory placed new demands on the swap device. Larger programs demanded larger swap areas, and constant paging generated more transfers to and from the swap device. To gain higher performance from the disks for paging, 4BSD uses swap areas on multiple disks. A single logical swap device is interleaved among the available swap partitions on the disks. The devices are interleaved on *dmmax*-block boundaries so that a maximally sized allocation will never span devices. *Dmmax* has a value that depends on the virtual-memory limit configured for the system; it ranges from 0.5 to 2 Mbyte. The swap devices to be used are configured into the kernel ahead of time; the size of each is computed during autoconfiguration. Swap areas need not be the same size; the interleaving is computed using the maximum size of the available devices. The ordering of the swap blocks on the interleaved devices is illustrated in Fig. 5.9; each square represents *dmmax* blocks, with virtual block numbers increasing in the order shown.

Figure 5.9 Swap-block interleaving.

swap-block numbers		12		14
		9	10	11
		6	7	8
		3	4	5
		0	1	2
swap device	0	1	2	

As the system must have at least one swap device available immediately, the first swap device is enabled at boot time; the virtual swap blocks that map to it are placed in a swap-resource map. The other swap devices are enabled by a *swapon* system call at system startup, and their sections of the interleaved swap device are made available at that time.

Swap space for each process segment is allocated when the segment grows. The resource allocator does not return swap blocks that span devices because of interleaving. Thus, individual chunks for a segment map to a single disk device. The use of interleaving rather than concatenation of the swap devices tends to distribute the allocation of space evenly on the swap devices so as to spread the transfers among the disks used for swapping and paging.

5.6 Per-Process Resources

In Chapter 4, we discussed several of the components of a UNIX process. Each process is described by a *proc* structure, which is allocated from an array of such entries in the kernel data segment, and by a user structure, which is allocated dynamically. In this section, we examine the memory utilization of UNIX processes in 4.3BSD. We start by examining how a process's page tables describe the process's virtual memory. This presentation is followed by a discussion of the various types of PTEs that may exist during the lifetime of a process. Finally, the shared resources associated with shared text segments and the space allocated on secondary storage for each process are considered.

4.3BSD Process Virtual Address Space

In 4.3BSD on the VAX, the P0 region is used to hold the text segment of a process, followed by the data segment. The kernel stack and user structure are stored at the top of the P1 region, with the user stack growing downward below them. The set of page tables for the P0 and P1 regions for each VAX process are saved and restored during process-context switches. The placement of the kernel stack and user structure in the P1 region allows those objects to be placed at fixed addresses, and also allows them to be remapped quickly as part of the user address space at each context switch. The pages at the top of the P1 region that hold the user structure and kernel run-time stack are made read-only to the user process. One advantage of having these pages readable to the user is that the signal-trampoline code stored in the user structure is readily available (see Section 4.7).

Page Tables

As we saw in looking at the VAX memory-management architecture in Section 5.3, each process must have a set of page tables that describes the memory associated with the process. The two-level translation process implies that processes hold two types of resources: *user page tables* to map P0 and P1 regions, and a portion of the *system page table* to map the user page tables.

System map
Alternate mappings
Utility maps
I/O map
User page-table map

Figure 5.10 Layout of the system page table.

In 4.3BSD, the system page table is statically allocated as the concatenation of a number of smaller page tables (see Fig. 5.10). The static entries in the map are filled in at boot time, and others are filled in as they are used. The first section (*Sysmap*) maps the fixed part of the kernel code, data, and tables allocated at boot time. This section also maps the filesystem's buffer cache (see Section 7.5).

The second area of the system page table is divided into sections that are used by different parts of the kernel to map in the user structures of processes that are not currently executing. The third area contains maps for dynamic allocation of kernel memory and temporary access of main memory pages. The fourth section of the map is set up to address I/O space rather than main memory, so that device drivers and autoconfiguration procedures can access peripheral-device registers directly. The last piece of the system page table, the user page-table map, or *Usrptmap*, is used to map user-process page tables into the kernel's address space. Because of the difficulty in allocating additional physically contiguous memory to expand the system page table, the *Usrptmap* has a fixed size. The standard size is 32 pages, enough to map 4096 pages of user page tables, which in turn map 524,288 pages of memory. A standard configuration thus supports page tables for resident processes whose virtual memory sizes total up to 256 megabytes.

A process needs several contiguous system PTEs to map its own page tables. The allocation of *Usrptmap* entries is managed with a *resource map* called *kernel-map*. Resource maps are ordered arrays of <base, size> pairs describing the free segments of a resource (see Fig. 5.11). A segment of a given size is allocated from a resource map by *rmalloc()*, using a first-fit algorithm, and is subsequently freed with *rmfree()*. The kernelmap is initialized at boot time to contain all the *Usrptmap* entries. An index into the kernelmap at which space has been allocated is used both as the index into the array of PTEs and as the index into the

Figure 5.11 A kernel resource map.

(base)	0	8	16	29	36	42	48
resource:		in use		in use		in use	
(size)	8	8	13	7	6	6	15

resource map: <0,8>, <16,13>, <36,6>, <48,15>

corresponding array of virtual addresses that corresponds to these pages. The P0 and P1 page tables are placed in adjacent areas using one set of kernelmap entries.

Reserving an area of the user page-table map provides a process with a virtual-address range in which to store its page tables. However, pages of physical memory still must be assigned to this virtual-address range. These pages are allocated with either *vmemall()* or *memall()*, both of which were described in Section 5.4.

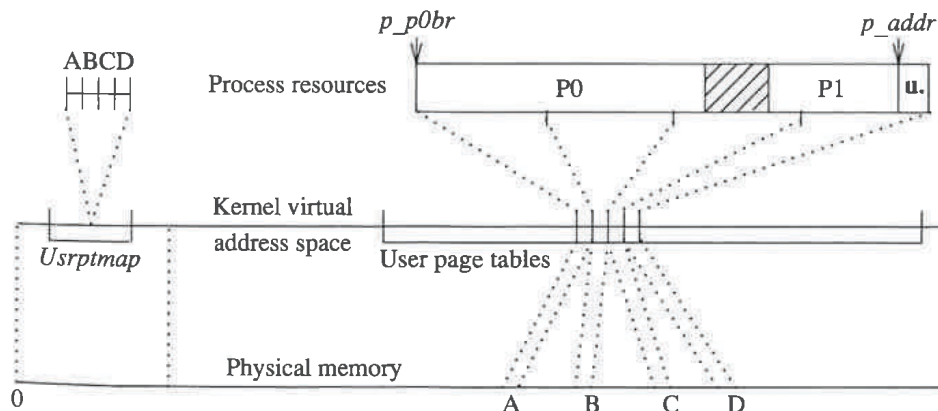
Note that all a process's user page tables are allocated during this procedure, even though the corresponding system PTEs could have remained invalid. 4BSD does this because it does not support paging of the page tables; it always allocates memory for all a process's page tables. This policy is a holdover from previous versions of the system, but has become rather costly as the maximum size of a process has grown over time.

Figure 5.12 shows the arrangement of the page tables for a process. The page tables for the process illustrated occupy four pages of memory, which are mapped by four kernel PTEs in *Usrptmap*. The first section of the process's page tables is the P0 page table, with P0BR pointing to its start. The value of P0BR is stored in the PCB for use by the VAX hardware, and is also stored in the proc structure as *p_p0br*. The P1 page tables are placed after the P0 tables, ending on a page boundary. An additional pointer into this area is maintained; the proc structure contains a field, *p_addr*, that points to the first PTE for the user structure (near the end of the P1 map). A gap between the two sections allows for a small amount of expansion in place. Expansion is complicated; it is dealt with in Section 5.9.

Types of Page-Table Entries

When a process is created, its address space is empty, so its PTEs are set up with the valid bit off. As the process executes, memory references to the text segment cause page faults that result in pages being brought in from the

Figure 5.12 Layout of page tables for a process.



associated object file image that resides on disk. These page faults are termed *fill-on-demand page faults*, and, thus, the associated PTEs in this initial state are called *fill-on-demand PTEs*.

When the valid bit is off in a PTE, only the protection field is examined by the hardware. The other parts of the entry may be used in any way by software. 4.3BSD takes advantage of this fact by storing precomputed information in the unused part of the PTE that will be needed during the processing of an initial page fault. All user-process PTEs are initialized in this fashion. As there are several possible reasons for the valid bit to be turned off later in execution, a software-only bit is defined to show whether a page is still a fill-on-demand page. If so, the alternate interpretation of the PTE information is still valid.

The structure definition for a normal PTE, *struct pte*, is given in Fig. 5.13. The fill-on-demand PTE, *struct fpte*, is shown in Fig. 5.14. The valid bit and protection bits are identical in both structures, as required by the hardware. The fill-on-demand bit is also common, and distinguishes which of the two structures should be used to interpret a given PTE.

The rules for distinguishing PTEs are summarized in Table 5.2. In the next four paragraphs, we describe the PTEs in each of these states.

A fill-on-demand page may be of one of two types: either fill-from-file, or fill-with-zeros. Text and initialized data pages are filled from the executable text image file, whereas uninitialized data (*bss*), dynamically extended heap, and stack areas are filled with zeros. These types are distinguished by the *pg_fileno* field in the *fpte* structure, which has a value of either PG_TEXT or PG_ZERO. When a page should be filled from the text file, the precomputed disk block number of the appropriate page will be found in the *pg_blkno* field and the disk device number will be available from the text-table entry. These values are computed all at once, at the time the page tables are initially created for a shared text file. This precomputing of the block numbers is done because the calculation usually requires many accesses to secondary storage and thus is relatively slow. Precomputation guarantees that all the necessary filesystem structures will be found in the buffer cache after their first access.

When a page is valid, it is assumed to be resident in main memory. Valid PTEs always contain a nonzero page number, which is the physical-memory page number of the page to which this address is mapped.

Figure 5.13 Normal page-table entry (*struct pte*).

```
struct pte {
    unsigned int    pg_pfnnum:21, /* page frame number */
                   :4,
                   pg_fod:1,     /* fill-on-demand */
                   pg_m:1,       /* modified */
                   pg_prot:4,    /* protection */
                   pg_v:1;       /* valid */
};
```

```

struct fpte {
    unsigned int  pg_blkno:24, /* filesystem block number */
                  pg_fileno:1, /* fill-from-text or with-zeros */
                  pg_fod:1,    /* fill-on-demand */
                  :1,
                  pg_prot:4,    /* protection */
                  pg_v:1;       /* valid */
};

```

Figure 5.14 Fill-on-demand page-table entry (*struct fpte*).

An invalid page with a nonzero page number recorded in its PTE has a page of main memory allocated to it. In this situation, one of the following is true:

1. The page is filling from the disk: A page-in operation is in progress.
2. The page is in transit to the disk: A page-out operation is in progress.
3. A reference bit on the page is being simulated.

The last case is peculiar to the VAX. When the page-replacement algorithm would clear the reference bit on a page (if it were present), it clears the valid bit instead. After the valid bit is cleared, the process cannot use the page without causing a page fault. The page-fault service routine will note that the page is otherwise valid, and that this fault serves only to note that the page is still in use. The page-fault service routine can simply restore the valid bit, marking the page as both valid and *referenced*. Section 5.12 discusses the operation of the page-replacement algorithm in detail.

The last possible state for a PTE is that where both valid and fill-on-demand bits are clear, and the page number is zero. This state represents a PTE whose page has been created and brought into memory, but whose page has since been moved to swap space.

Table 5.2 Types of page-table entries.

Valid (<i>pg_v</i>)	Fill-on-demand (<i>pg_fod</i>)	Page number (<i>pg_pfnum</i>)	Structure type, usage
0	1	(none)	<i>struct fpte</i> ; fill-on-demand page
1	0	nonzero	<i>struct pte</i> ; valid, resident page
0	0	nonzero	<i>struct pte</i> ; invalid, resident page
0	0	zero	<i>struct pte</i> ; invalid, nonresident page
1	1	(none)	unused

Modified Pages

The last field in the PTE that is used for valid pages is the modified bit. This bit is set on the VAX by the hardware when a translation is done for the page during a write operation. When a page marked as modified is to be moved from memory, the page must be saved in swap space before it is freed. When a page is first created and filled, the modified bit is set to force the page to be saved the first time it is released. Text pages could be refilled from the filesystem, but their filesystem location is overwritten by the physical page number. It is faster to save them on swap space than to recalculate the disk block location, as the latter may require reading indirect blocks (see Section 7.2).

Text Page Tables

The only memory shared among processes in 4.3BSD is the instruction area of a pure text-executable program (except for *vfork*, which is described in Section 5.7). On the VAX, the most efficient way to implement this sharing is to have one set of page tables for each shared text segment, and then to have each process sharing a text segment simply reference these page tables. This was not done, however, because it would have required the text section to start and end on 64-Kbyte boundaries, resulting in an incompatible user-visible change for binary sites. Instead, processes sharing a text segment maintain individual copies of page tables that reference the physical pages of a text segment, and the system ensures that these tables remain consistent by propagating changes that result from loading, invalidating, or paging out text pages.

Swap Space

In a virtual-memory operating system, programs may run with less than their full address space resident in physical memory. But the system must be able to obtain or create the contents of any section of the address space at any time. With demand paging, a process initially has nothing in its address space. Some portions of the address space are mapped to the executable image in a file, whereas other portions are marked to be filled with zeros on first reference. Later, some pages that have already been created and used may need to be removed from memory to make room for other pages. The pages that are removed must be saved on secondary storage if their contents cannot be reproduced. The 4BSD virtual-memory system never attempts to reproduce the contents of a page once that page has been brought into memory (for example, by paging the page from the executable image again if both the file and the page are unmodified). Instead, such pages are stored in swap space. Swap space uses reserved areas of the disks. These areas contain no filesystem structures, and are usually 16 or 32 Mbyte per area on one or more disks. To avoid running out of swap space at an inconvenient time, the system allocates enough swap space for each program, so that all a program's pages can be written to swap space if necessary. The decision to preallocate swap space was reasonable when the system was originally designed. VAX systems of the day

generally had no more than 4 Mbyte of main memory, but usually had several disks totaling hundreds of Mbyte. However, memory sizes of 16 to 128 Mbyte have become more common, while disk sizes have grown much less rapidly.

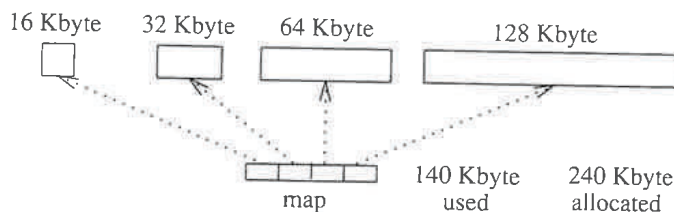
The allocation and mapping of swap space for a process is described by a data structure called a *swap map*. Each process has two swap maps, one describing the swap area for its data and heap segment (P0 space excluding text) and one describing its stack segment. A swap map describing the text segment of a shared text image may also be referenced by a process. The shared text swap map is maintained separately from the per-process swap maps, and is described later.

Each per-process area may grow dynamically and independently, and may occupy a wide range of sizes, from a few Kbyte to many Mbyte. If a swap map used fixed-sized blocks, either large processes would require large maps, or small processes would have to overallocate, losing much space because of internal fragmentation. The small-process problem could be solved by reducing the last swap block to the actual size required. However, such a reduction would make growth of a segment expensive, as it might require a disk-to-disk copy of a small swap block to a larger one.

The solution to this problem in 4.3BSD is the use of a fixed-size swap map that contains swap addresses for varying sizes of blocks. Figure 5.15 shows the swap map for a 140-Kbyte process. The size of each block is determined by its index in the map. The first block is of size *dmmin*; standard 4.3BSD configurations set *dmmin* to 32 sectors (16 Kbyte). Succeeding blocks are each larger than their predecessor by a factor of two until they reach size *dmmax*. Each block used by a process is fully allocated, so growth to the end of the block is simple. Growth past the end of the last block is done by allocation of a new block. Small processes thus consist entirely of small swap blocks, minimizing fragmentation, whereas large processes can still be described by a map of moderate size. The actual process-segment swap map, called a *dmap* structure, contains the amount of swap space allocated to the segment, the amount currently used, and a fixed-size array of swap block addresses for the blocks that are allocated. The user structure of each process holds maps for the data portion of P0 space (*u_dmap*) and the stack portion of P1 space (*u_smap*).

Swap space for the text segment is handled differently; only one copy needs to be stored on disk, and its size is constant. The text structure holds an array of

Figure 5.15 The *dmap* structure.



block pointers to map the swap image. Each block but the last has a fixed size, *dmtext*, ordinarily 512 Kbyte; the last block holds just the number of 1-Kbyte clusters needed for the remainder of the text image.

5.7 Creation of a New Process

UNIX processes are created with a *fork* system call. The *fork* is usually followed shortly thereafter by an *exec* system call that overlays the virtual address space of the child process with the contents of an executable image that resides in the filesystem. The process then executes until it terminates by exiting, either voluntarily, or involuntarily by receiving a signal. In the next few sections, we trace the management of memory resources used at each step in this cycle.

A *fork* system call duplicates the address space of an existing process, creating an identical child process. *Fork* is the only way that new processes are created in UNIX (except for its variant, *vfork*, which is described later). The 4.3BSD implementation of *fork* duplicates all the resources of the original process and copies that process's address space.

The virtual-memory resources of the process that must be allocated for the child include the *proc* structure, the secondary storage used to back the process, the kernel page tables and user page tables, the user structure, and the kernel stack. Main memory is allocated for many of the pages of the new process (those that are not still fill-on-demand in the parent process), although it may need to be paged out by the time the process first runs. The general outline of the implementation of a *fork* is as follows:

- Allocate secondary storage for the child process
- Allocate a *proc* entry for the child process and fill it in
- Allocate page tables for the child process
- Allocate a new user structure, copying the current one to initialize it
- Duplicate the address space, copying fill-on-demand PTEs and duplicating the contents of all other pages
- Arrange for the return to the child process, to distinguish it from the parent process

The allocation and initialization of the *proc* structure and the arrangement of the return value were covered in Chapter 4. The remainder of this section discusses the other steps involved in duplicating a process.

Duplicating Kernel Resources

The first resource to be allocated when an address space is duplicated is the swap area. Swap space is allocated first, because it is the resource most likely to be unavailable. As the existing process already has swap maps for its own swap

space, the new allocation must be recorded in *shadow swap maps* in the user structure until they can be installed in the child-process user structure. These swap maps, *u_cdmmap* and *u_csmap*, are also used during *exec*; their treatment is described later. The kernel allocates swap space by zeroing the shadow maps, then expanding these maps to the full size of the process data and stack segment sizes. The process then holds enough swap space for two complete copies of itself, one copy of which has no contents.

The next step in *fork* is to allocate and initialize a new *proc* structure. This operation must be done before the address space of the current process is duplicated, because it records state in the *proc* structure. To duplicate the address space in the new process, the kernel must allocate page tables and pages of main memory. Allocation of the page tables is done by *vgetpt()*, which allocates system PTEs from *kernelmap* to map the user page tables, then allocates memory pages to contain the user page tables from the free memory pool with *vmemall()*. If space is not available in the system page table, the allocation request will cause the process to block. Lack of space in the system page table is usually due to fragmentation. When this occurs, the system resorts to the *swapping* of processes as a means of reclaiming resources; this mechanism is described in Section 5.13.

Between the time that space is allocated in the system page table and the time that user page tables are allocated, the parent process is locked against swapping to avoid deadlock. The child is in an inconsistent state, and cannot yet run or be swapped, and the parent is needed to complete the duplication of page tables. To ensure that the child process is ignored by the scheduler, the kernel sets the process's state to *SIDL* during the entire *fork* procedure.

With the virtual-memory resources allocated, the system sets up the kernel- and user-mode state of the new process, including the hardware memory-management registers and the user structure. Creating a copy of the user structure requires that a consistent copy of the VAX process control block be obtained. The parent process does this by simulating a context switch to itself:

```
resume (pcbb(u.u_procp));
```

Then, so that the current and the new user structures can be accessed simultaneously, the new user structure is mapped into a special window in kernel virtual memory using a special page map named *Forkmap*. With both parent and child user structures addressable in the kernel's virtual address space, the current user structure can then be copied to the new one, and the virtual-memory section of the new user structure can be updated to reflect the child's page tables and swap-space allocation.

Duplicating the User Address Space

The remainder of the user address space may be duplicated once the kernel state of the child process has been created. The text portion is easy; the reference count on the text structure is incremented to show that there is another process sharing it, and the page tables are copied from a current process using that text segment.

Finally, the data and stack areas are duplicated one page at a time. For pages that are still fill-on-demand, only the PTEs need to be duplicated. The kernel copies all other pages by allocating memory for the new process and copying the data from the current process to the new memory page. The copy is done from a user virtual address to a physical address, because the new address space is not accessible to the parent process. The machine-language routine *copyseg()* does this copy by using a reserved system map entry to map in the destination page before copying the data. If the source page is not resident in memory, the copy will cause a page fault; the fault is serviced normally, and the copy proceeds on return. (Note that this situation is one of the few places in the kernel where a fault in kernel mode is not treated as a system failure.) The new PTE is validated and marked as modified, as the corresponding swap block does not yet have the correct contents.

Implementation Issues

When large processes fork, copying the entire user address space is expensive. All the pages that are on secondary storage must be read back into memory to be copied. If there is not enough free memory for both complete copies of the process, this memory shortage will cause the system to begin paging to create enough memory to do the copy (see Section 5.12). This may result in parts of the parent and child processes being paged out, as well as in the paging of parts of unrelated processes. Other ways to create processes without this overhead have been considered.

One technique that was considered for inclusion in 4.2BSD, but that was not implemented, is called *copy-on-write*. Rather than each page of a process that forks being copied, each process resulting from the fork is given references to the same physical pages. The page tables are changed to prevent either process from modifying a shared page. Instead, when a process attempts to modify a page, the kernel is entered with a protection fault. On discovering that the fault was caused by an attempt to modify a shared page, the kernel simply copies the page and changes the protection field for the page to allow modification once again. Only pages modified by one of the processes need to be copied. Because processes that fork typically overlay the child process with a new image with *exec* shortly thereafter, this technique can significantly improve performance.

Copy-on-write was not part of the original design of the virtual-memory system for various pragmatic reasons. It requires that reference counts be maintained on a per-page basis—which is most important for pages resident on secondary storage. More important, at the time, microcode problems with one model of the VAX made it unclear whether copy-on-write could be implemented reliably. As a result, the *vfork* system call, described in the next section, was included instead.

Creating a New Process Without Copying

A different scheme used to avoid the memory copy implied in a *fork* eliminates the copy altogether. When a process (such as a shell) wishes to start another program, it will generally *fork*, do a few simple operations such as redirecting I/O descriptors and changing signal actions, and then *start* the new program with an

exec. In the meantime, the parent shell suspends itself with *wait* until the new program completes. For such operations, it is not necessary for both parent and child to run simultaneously, and therefore only one copy of the address space is required. This frequently arising situation led to the implementation of the *vfork* system call. Although it is extremely efficient, *vfork* has peculiar semantics and is generally considered to be an architectural blemish.

Most of the implementation of *vfork* is shared with *fork*. The major difference is that, instead of copying the address space for the child, the parent simply passes its address space to the child and suspends itself. The child process needs to allocate only new proc and user structures, receiving everything else from the parent. The child process returns from the *vfork* system call with the parent still suspended. The child does the usual activities in preparation for starting a new program, then calls *exec*. Now the address space is passed back to the parent process, rather than being abandoned as in a normal *exec*. Alternatively, if the child process encounters an error and is unable to execute the new program, it will *exit*. Again, the address space is passed back to the parent instead of being abandoned. This sequence of events is illustrated in Fig. 5.16.

The child process in a *vfork* is created initially with just a user structure and enough page tables to map that structure. The parent then exchanges address spaces with the child. Most of the work is in exchanging pointers, maps, and counters in the proc and user structures that describe the address space. The exchange of the page tables is complicated, because the active kernel stack is contained in the user structure that is being passed. The transfer is done in the context of the parent process by mapping the child's user structure using the *Forkmap*, then copying the parent's PTEs for the user structure into the child's page tables. When the page tables are really swapped by changing the hardware page-table base registers, the user structure for the parent retains the correct mapping. The page map for the child's user structure is restored from the copy in *Forkmap*.

After passing its resources to the child, the parent process marks its own process-table entry with the flag SNOVM (has no virtual memory resources), so that it will be ignored during page reclamation, and also marks the child's entry with SVFORK. The parent then blocks until the child finishes its use of the parent's resources. When the child calls either *exec* or *exit*, it clears the SVFORK flag, awakens the parent, and sleeps until the parent has taken back its address space. The reverse transfer is done in an identical fashion to the first transfer, with care taken to retain the correct mapping for the user structure. When the parent has completed the transfer, it marks the child with the flag SVFDONE and awakens it. When the child sees that the transfer is complete, it continues with its system call; in either case, it has fewer resources to release than in a normal process.

This description of *vfork* shows that neither the pages of the address space nor the page maps themselves need to be copied. *Vfork* is likely to remain more efficient than copy-on-write or other schemes that must duplicate the process' virtual address space. The architectural quirk of the *vfork* call is that the child process may modify the contents and even the size of the parent's address space while the child has control. Although modification of the parent's address space

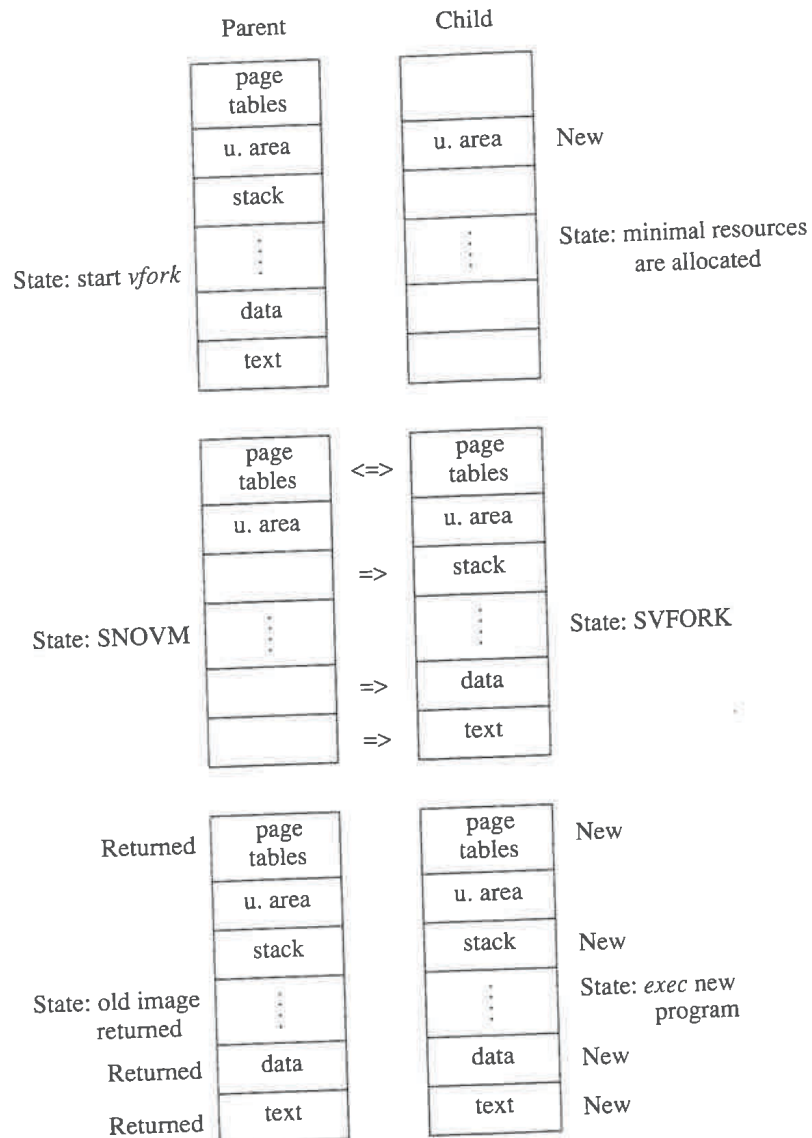


Figure 5.16 Operation of *vfork*.

is bad programming practice, some programs have been known to take advantage of this quirk.

5.8 Execution of a File

The *execve* system call was described in Sections 2.4 and 3.6; it replaces the address space of a process with the contents of a new program obtained from an executable file. During an *exec*, the target executable image is validated, then the

arguments and environment are copied from the current process image into a temporary area.¹

To perform an *exec*, the system must allocate resources to hold the new contents of the virtual address space, set up the mapping for this address space to reference the new image, and release the resources being used for the existing virtual memory. This procedure is complicated by the possibility of the process being swapped to secondary storage if resources, such as the system page tables, are in short supply.

The first step is to allocate swap space for the new data and stack sections. *Exec()* does this allocation without first releasing the currently assigned space, because the system must always have a place on secondary storage to which the process can be swapped (see Section 5.13). If the system were to release the current swap space, and the swap allocation failed, the *exec* would be unable to return to the original process. The duplicate swap space that is allocated is recorded in *shadow swap maps* in the process' user structure, and the address space and virtual-memory resources of the current process are then freed as though the process were exiting; this mechanism is described later.

Now, the process has only a user structure and kernel stack, plus page tables that map only the user structure. The shadow swap maps that hold the new swap area are moved to their normal locations, and resources are arranged for the new text, data, and stack sizes of the process. This arrangement is done by expansion of the page tables to the new size and updating of the hardware registers. The PTEs for the stack segment are set to be fill-with-zeros and, if the image about to be executed is not demand-loaded (an uncommon case), the initialized data are read into the data space, faulting in the required pages as the read proceeds.

The kernel sets up the text portion of the new address space by locating the text structure for the text file if that file is already in memory, or by allocating and initializing a new text structure if the text file is not in memory. When a new text structure is being set up, swap space is allocated for the text segment and for a swap image of the page table for the text. Otherwise, the process is added to the list of processes sharing the text. The text page tables are initialized in one of three ways:

1. If another process using this text is present in memory, the page tables for the text region are simply copied from those of the other process.
2. Otherwise, if a copy of the page tables for text are on swap space, the swap copy is read into place.
3. If neither of these shortcuts is possible, the page tables are set up as fill-from-file.

The third case represents the largest overlap of information between the filesystem and the virtual-memory system; text PTEs are initialized with the disk

¹ A section of the first swap device, called *argdev*, is reserved for argument processing. Enough disk blocks to hold a maximal argument list are allocated and arguments are copied into I/O buffers identified with the disk blocks for that area. In normal operation, these buffers are never written to disk, as their contents are retrieved before any write operation occurs.

block number for each page cluster. For each filesystem block that is mapped, the disk block number must be found using *bmap()*, and must be recorded in the block numbers for the page clusters that map the block. This mapping is an expensive operation, since *bmap()* may need to read through indirect blocks to find the location of some of the filesystem blocks. It is thus more efficient to do all these mappings at one time, so that indirect blocks will remain in the buffer cache during the operation. (See Chapter 7 for more of the details of filesystem operation.) Following the initialization of the text structure and page tables, the page tables for initialized data are set to fill-from-file.

No further operations are needed to create a new address space during an *exec* system call; the remainder of the work comprises copying the arguments and environment out to the top of the new stack. Initial values are set for the registers: The program counter is set to the entry point and the stack pointer is set to point to the argument vector. The new process image is then ready to run.

5.9 Change Process Size

A process may change its size during execution by explicitly requesting with the *sbrk* system call that the size of the data (heap) region be changed. Also, the stack segment will be expanded automatically if a protection fault is encountered because of growth of the stack below the end of the stack region. In either case, the size of the process address space must be changed. Hence, the size of the appropriate page table and swap map must be changed. If the change is to reduce the size of the data segment, the operation is easy: Any memory allocated to the pages that will no longer be part of the address space is freed, and the excess PTEs are cleared. Future references to these addresses will result in protection faults, as access is disallowed when the protection field is zero.

More commonly, the direction of change is to enlarge one of the two process segments, data or stack. New pages are marked fill-with-zeros, as there are no contents initially associated with new sections of the address space. The first step is to check whether the new size would violate the size limit for the process segment involved. If the new size is in range, the swap map for the segment is enlarged as needed. Recall that the swap maps for these segments are allocated in blocks of fixed sizes according to their indices in the map, and that both the current size and current allocation are maintained. If the current allocation is large enough for the new size, only the size needs to be changed; no new allocation is necessary. If the size has exceeded the current allocation, new full-sized blocks are added to the map in succeeding slots until enough space has been allocated. This process is identical to the one that occurs when a process is created by *exec*.

Once enough swap space has been obtained, the address space is adjusted through alteration of the region sizes in the *proc* and *user* structures, and then computation of the new size of the page tables. The size of the page tables is always rounded up to the end of the next page cluster, with any unused entries between the P0 and P1 page tables. There may be enough room for the additional entries. If not, the page tables for the process must be expanded. The page tables

are always expanded by allocation of a complete new section of the system page table and copying of the page table PTEs; for simplicity, the growth is never done in place by allocation of adjoining free entries. As discussed in Section 5.6, a process page table requires several contiguous system PTEs allocated from the kernel map, as well as physical memory mapped by the system PTEs.

Figure 5.17 indicates how a new set of system PTEs is allocated in the user page-table map so that enough pages can be mapped for the new page map. Rather than allocating new kernel memory for the user page tables and copying the tables, the kernel maps the page-table pages into the new area by copying the system PTEs in the user page-table map. The pages containing only stack or u-dot page tables are moved to the end of the new area, and the remaining pages are moved to the beginning of the new area, as shown in step 1. The new page(s) between these sections are allocated and are cleared so that unused portions will have their protection fields set to zero. Any stack (P1) entries that were sharing a page with the P0 map are copied to their correct locations at the end of the new pages, as shown in step 2. During this period, the user page tables are doubly mapped at both the new and old addresses; the kernel is running on a stack in P1 space, and it is important that the stack page tables not become invalid during this manipulation. Once the new user page tables are ready, the hardware and software page-table base and length registers are updated, and the old system PTEs can be freed. The new PTEs mapping the portion of the address space that was specifically requested are initialized to be read-write, fill-with-zeros; the other new entries are left uninitialized until a new request requires their use.

If either the system page-map allocation from the kernel map or the memory allocation for the new page-table pages fails, the kernel swaps out the process to await enough page-map space. Because the new sizes are already set up (in the process structure), when the process is swapped in it will have the correct size. The growth process will complete with the initialization of the new PTEs. A process would therefore be required to swap to expand its page tables if it grew large enough to require one-half of the system page table.

Figure 5.17 Page-table expansion.

