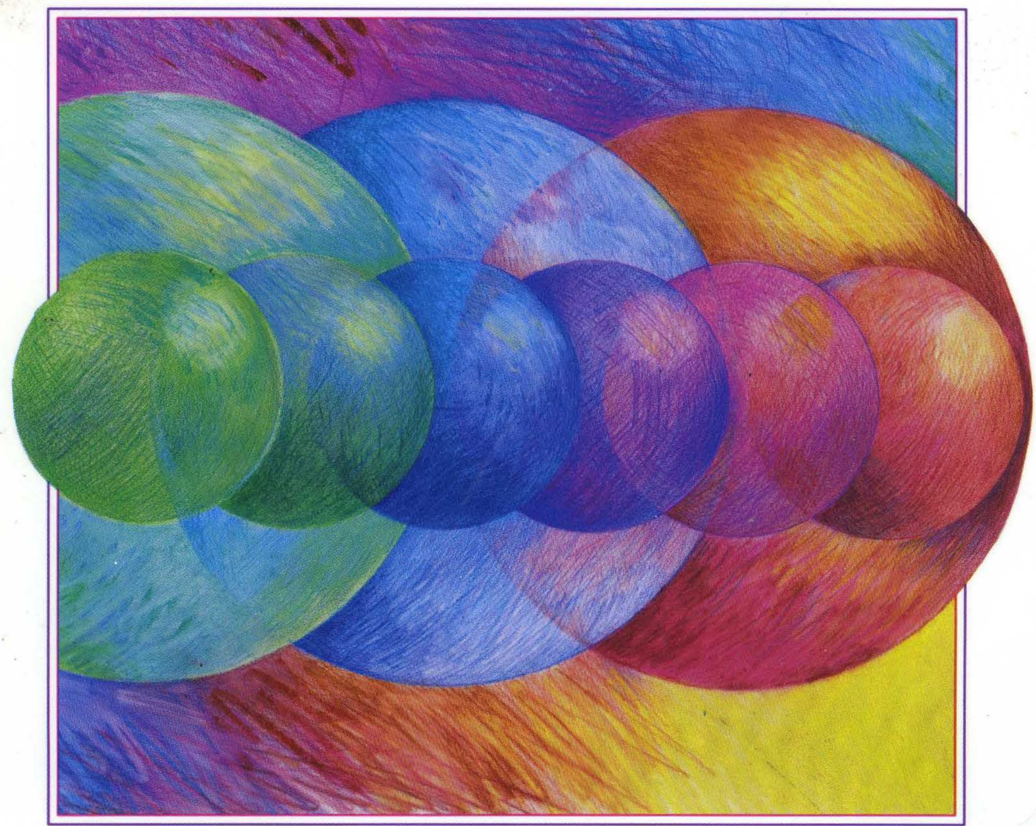
Rel. 1.2

OSF/1™ Operating System

Release 1.2

Design of the OSF/1 Operating System



OPEN SOFTWARE FOUNDATION

Design of the OSF/1 Operating System

Design of the OSF/1 Operating System

Release 1.2

Open Software Foundation

P T R Prentice Hall, Englewood Cliffs, New Jersey 07632

Cover design
and cover illustration: **BETH FAGAN**

This book was formatted with troff.

Published by P T R Prentice-Hall, Inc.
A Simon & Schuster Company
Englewood Cliffs, New Jersey 07632

The information contained within this document is subject to change without notice.

OSF MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

OSF shall not be liable for errors contained herein, or for any direct or indirect, incidental, special or consequential damages in connection with the furnishing, performance, or use of this material.

Copyright ©1993 Open Software Foundation, Inc.

This documentation and the software to which it relates are derived in part from materials supplied by the following:

- © Copyright 1987, 1988, 1989 Carnegie-Mellon University
- © Copyright 1985, 1988, 1989, 1990 Encore Computer Corporation
- © Copyright 1985, 1987, 1988, 1989 International Business Machines Corporation
- © Copyright 1988, 1989, 1990 Mentat Inc.
- © Copyright 1987, 1988, 1989, 1990 SecureWare, Inc.
- This software and documentation are based in part on the Fourth Berkeley Software Distribution under license from The Regents of the University of California. We acknowledge the following individuals and institutions for their role in its development: Kenneth C.R.C. Arnold, Gregory S. Couch, Conrad C. Huang, Ed James, Symmetric Computer Systems, Robert Elz © Copyright 1980, 1981, 1982, 1983, 1985, 1986, 1987. Regents of the University of California.

All rights reserved.

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

ISBN 0-13-202813-1

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

THIS DOCUMENT AND THE SOFTWARE DESCRIBED HEREIN ARE FURNISHED UNDER A LICENSE, AND MAY BE USED AND COPIED ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE AND WITH THE INCLUSION OF THE ABOVE COPYRIGHT NOTICE. TITLE TO AND OWNERSHIP OF THE DOCUMENT AND SOFTWARE REMAIN WITH OSF OR ITS LICENSORS.

Open Software Foundation, OSF, the OSF logo, OSF/1, OSF/Motif, and Motif are trademarks of the Open Software Foundation, Inc.

UNIX is a registered trademark of UNIX System Laboratories, Inc. in the U.S. and other countries.

X/Open is a trademark of the X/Open Company Limited in the U.K. and other countries.

AT&T is a registered trademark of American Telephone & Telegraph Company in the U.S. and other countries.

BSD is a trademark of University of California, Berkeley.

DEC and DIGITAL are registered trademarks of Digital Equipment Corporation.

Ethernet is a registered trademark of Xerox Corporation.

X Window System is a trademark of the Massachusetts Institute of Technology.

Sun, Network File System, and NFS are trademarks of Sun Microsystems, Inc.

SMP, SMP+, and CMW+ are trademarks of SecureWare, Inc.

PostScript is a trademark of Adobe Systems Incorporated.

Apple, the Apple Logo, Macintosh, AppleTalk, ImageWriter, and LaserWriter are registered trademarks of Apple Computer, Inc. A/UX is a trademark of Apple Computer.

FOR U.S. GOVERNMENT CUSTOMERS REGARDING THIS DOCUMENTATION AND THE ASSOCIATED SOFTWARE.

These notices shall be marked on any reproduction of this data, in whole or in part.

NOTICE: Notwithstanding any other lease or license that may pertain to, or accompany the delivery of, this computer software, the rights of the Government regarding its use, reproduction and disclosure are as set forth in Section 52.227-19 of the FARS Computer Software-Restricted Rights clause.

RESTRICTED RIGHTS NOTICE: Use, duplication, or disclosure by the Government is subject to the restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 52.227-7013.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in paragraph (b)(3)(B) of the rights in Technical Data and Computer Software clause in DAR 7-104.9(a). This computer software is submitted with "restricted rights." Use, duplication, or disclosure is subject to the restrictions as set forth in NASA FAR SUP 18-52.227-79 (April 1985) "Commercial Computer Software-Restricted Rights (April 1985)." If the contract contains the Clause at 18-52.227-74 "Rights in Data General" then the "Alternate III" clause applies.

US Government Users Restricted Rights - Use, duplication, or disclosure restricted by GSA ADP Schedule Contract.

Unpublished - All rights reserved under the Copyright Laws of the United States.

This notice shall be marked on any reproduction of this data, in whole or in part.

Contents

Preface . . . xv

Audience . . . xvi

Applicability . . . xvi

Document Usage . . . xvi

Related Documents . . . xvii

Problem Reporting . . . xviii

Acknowledgements . . . xviii

Chapter 1. The OSF/1 Operating System . . . 1–1

1.1 UNIX Functionality . . . 1–2

1.2 Advanced Features . . . 1–3

1.3 What is OSF/1? . . . 1–4

1.3.1 Tasks and Threads . . . 1–4

1.3.2 Virtual Memory and Memory Management . . . 1–5

1.3.3 File Management . . . 1–5

1.3.4 Networking . . . 1–6

1.3.5 STREAMS . . . 1–7

1.3.6 Sockets . . . 1–7

1.3.7 XTI . . . 1–8

1.3.8 Internationalization . . . 1–8

1.3.9 Terminals . . . 1–9

1.3.10 Logical Volume Manager . . . 1–9

1.3.11 Program Loader . . . 1–10

1.3.12 Security . . . 1–11

1.3.13 Scalability and Dynamic Configuration . . . 1–11

1.4 The Future of the OSF/1 Design . . . 1–12

Chapter 2. Overview of UNIX Processes and the UNIX Kernel . . . 2–1

2.1 Process Address Spaces . . . 2–2

2.2 Process Management System Calls 2–3

2.3 Process States 2–4

2.4 Memory Management 2–5
2.4.1 Memory Management Techniques 2–6
2.4.2 The Transparency of Memory Management 2–7

2.5 Process Context and Context-Switching 2–8

2.6 The UNIX Kernel and Its Services 2–10
2.6.1 System Calls 2–10
2.6.2 Program Exceptions 2–11
2.6.3 Peripheral Device Activity 2–11
2.6.4 The Hardware Clock 2–12
2.6.5 Kernel Daemons 2–13

Chapter 3. Overview of the Mach Technology in OSF/1 3–1

3.1 Tasks and Threads 3–2
3.1.1 The task Data Structure 3–3
3.1.2 The thread Data Structure 3–3

3.2 The Mach Interprocess Communication Subsystem 3–4
3.2.1 Ports 3–4
3.2.2 Messages 3–6
3.2.3 Ports as Objects 3–8

3.3 Memory Objects 3–8

3.4 Mach Virtual Memory Management 3–9
3.4.1 Task Address Maps 3–10
3.4.2 Virtual Memory Objects and Memory Objects 3–11
3.4.3 VM Object Types 3–12
3.4.4 Memory Objects and Memory Managers 3–13
3.4.5 Management of Resident Pages 3–14
3.4.6 Physical Maps 3–16
3.4.7 Mach Virtual Memory Interfaces 3–17
3.4.8 Memory Managers and the External Memory Management Interface 3–19

Chapter 4. Processes: Structure and Management 4–1

4.1 Process States and Data Structures 4–1
4.1.1 The proc Structure 4–3
4.1.2 The user Structure 4–4

4.2 Allocation of proc Structures 4–5

4.3 The Process Management System Calls and Multithreaded Behavior 4–6
4.4 The Signal Facility 4–7
4.4.1 The Posting of Signals 4–8
4.4.2 Signal Delivery 4–9
4.4.3 The Signal System Calls 4–10
4.4.4 Implementation of the Signal Facility 4–11
4.4.5 The Exception Handling Facility 4–13
4.4.6 Signal Handlers 4–16
4.4.7 Unix System Calls, the U-area, and Interrupted System Calls 4–17

Chapter 5. The Scheduling Subsystem 5–1
5.1 Timesharing 5–2
5.1.1 The BSD Scheduler 5–2
5.1.2 The OSF/1 Scheduler 5–4
5.1.3 The Run Queue Data Structure 5–7
5.2 Thread Execution States 5–8
5.2.1 The Suspend Mechanism 5–9
5.2.2 Execution State and the Suspend Mechanism 5–10
5.2.3 The Event-Wait Mechanism 5–13
5.2.4 Execution State and the Event-Wait Mechanism 5–16
5.3 Scheduler Support for Parallel Applications 5–18
5.3.1 Processors and Processor Sets 5–19
5.3.2 Scheduling Hints 5–23
5.4 CPU-Usage Timer Support 5–25
5.4.1 OSF/1 Timers 5–26
5.4.2 The timer Data Structure 5–27

Chapter 6. The Virtual Memory Subsystem: Address Space Implementation 6–1
6.1 Address Maps and Address Map Entries 6–2
6.1.1 The vm_map Data Structure 6–3
6.1.2 The vm_map_entry Data Structure 6–4
6.1.3 Address Map Entries and the Page Fault Handler 6–7
6.2 Implementation of UNIX Process Address Spaces 6–8
6.3 The Optimization of Virtual Copy 6–9
6.3.1 Symmetric Copy-on-Write 6–10
6.3.2 Asymmetric Copy-on-Write 6–22
6.4 The Page Fault Handler and Copy-on-Write 6–26

6.5 Share Maps 6–27

6.6 Virtual Copy and Mach IPC 6–29

6.7 The Kernel's Address Space 6–29
6.7.1 Submap Implementation 6–30

6.8 Pmaps and the Pmap Module 6–32
6.8.1 The Pmap Functions 6–32
6.8.2 The Shootdown of Translation Lookaside Buffers 6–37

Chapter 7. The Virtual Memory Subsystem: Memory Management 7–1

7.1 Overview 7–2

7.2 The Vnode Pager 7–3
7.2.1 Paging Files 7–4
7.2.2 Page Clustering 7–5
7.2.3 Allocating Clusters in Paging Files 7–6
7.2.4 Vnode Pager Memory Objects 7–7

7.3 Cluster Paging Operations on Temporary Data 7–8

7.4 The Page Replacement Mechanism 7–8
7.4.1 Pageout of Data Managed by External Memory Managers 7–9
7.4.2 Pageout of Data Managed by the Vnode Pager 7–11

7.5 The Page Fault Handler and Pagein of Clusters 7–12

7.6 The Swapping Mechanism 7–13
7.6.1 Swapping Policy 7–14
7.6.2 The Thread and Task Swappers 7–15

7.7 External Memory Managers 7–17
7.7.1 Example of an External Memory Manager: A Simple Shared Memory Server 7–17
7.7.2 The External Memory Management Interface 7–19

Chapter 8. The OSF/1 Program Loader 8–1

8.1 Conceptual Background 8–2
8.1.1 Linking 8–3
8.1.2 Shared Libraries 8–4
8.1.3 The OSF/1 ld Command 8–4
8.1.4 Object Files and Object File Formats 8–4

8.2 Overview of the Program Loading Architecture in OSF/1 8–6
8.2.1 The Architecture of exec( ) in OSF/1 8–6

8.2.2 The Loader's Architecture 8–8

8.3 The Symbol Resolution Policy 8–9
8.3.1 Using Packages 8–10
8.3.2 Package Tables 8–11

8.4 The Loader Context 8–12
8.4.1 Module Records 8–14
8.4.2 Building the Known Modules List 8–15

8.5 The Loader Switch and Format-Dependent Managers 8–17
8.5.1 Format-Dependent Routines 8–17

8.6 Address Space Management 8–18
8.6.1 Absolute and Relocatable Regions 8–18
8.6.2 Base Addresses and Virtual Addresses for a Region 8–19
8.6.3 Context-Specific Allocation Procedures 8–19
8.6.4 Typical Loader Address Space Usage 8–20

8.7 Kernel Space Loading 8–21

8.8 Preloading, Installing Libraries, and the Global Data File 8–22

8.9 Dynamic Format Manager Loading 8–23

8.10 Unloading 8–24

8.11 Application Interface to the Loader 8–24

8.12 The Loader and Security 8–25

Chapter 9. Loading and Configuring Dynamic Subsystems 9–1

9.1 Overview: Loading and Configuring Dynamic Subsystems 9–2

9.2 Configuration and Kernel Tables 9–2

9.3 The Configuration Manager 9–3

9.4 Interrupt Handling 9–4
9.4.1 The locore.s Module 9–4
9.4.2 The Interrupt Dispatcher 9–5

9.5 Device Driver Configuration 9–6

9.6 Configuration of File Systems 9–8

9.7 Dynamic Loading and Configuring of System Calls 9–9
9.7.1 Selecting the System Call Number 9–9

9.8 Boot-Time Subsystem Configuration 9–10

Chapter 10. Internationalization Subsystem 10–1

10.1 Locales 10–2
10.1.1 Languages and Code Sets 10–2
10.1.2 Collating Conventions 10–4
10.1.3 Character Classification 10–6
10.1.4 International Date and Time Formats 10–6
10.1.5 International Numeric and Monetary Formats 10–8

10.2 Internationalization Subsystem Design 10–8

10.3 Application Programming Interface 10–9

10.4 Message Subsystem 10–11

10.5 OSF/1 Code Sets 10–12
10.5.1 EUC Code Sets 10–14
10.5.2 SJIS Code Set 10–14

10.6 The iconv Conversion Subsystem 10–15

10.7 Terminal Device Support for Internationalization 10–16
10.7.1 Initialization of Terminal Lines 10–18
10.7.2 Reconfiguring Terminal Lines 10–19

Chapter 11. File Management 11–1

11.1 Descriptor Management 11–3
11.1.1 Data Structures 11–4
11.1.2 Synchronization on Descriptors 11–7

11.2 Virtual File System Management 11–8
11.2.1 An External View of the File System Tree 11–9
11.2.2 The VFS Switch 11–10
11.2.3 Internal Representation of Mounted File Systems 11–10
11.2.4 Pathname Translation from Name to Vnode 11–12

11.3 Vnode Management 11–17
11.3.1 The Contents of a Vnode 11–17
11.3.2 The Free List and Cache 11–20
11.3.3 The Life Cycle of a Vnode 11–21
11.3.4 File Locking 11–24
11.3.5 Special Files 11–24
11.3.6 The Buffer Cache 11–27

11.4 The File System Layer 11–30
11.4.1 NFS 11–30
11.4.2 UFS 11–32

11.4.3 The System V File System 11–35
11.4.4 File System Security Extensions 11–38

Chapter 12. Sockets 12–1

12.1 The Socket Framework 12–2

12.2 The Socket Programming Interface 12–2

12.3 Domains and Protocols 12–3
12.3.1 Domain Overview 12–3
12.3.2 The domain Structure 12–4
12.3.3 Adding and Deleting Protocols 12–5

12.4 The socket Data Structure 12–6

12.5 Scheduling Network Activity 12–8
12.5.1 Event Management 12–9
12.5.2 The netisr Structure 12–10
12.5.3 Packet Processing 12–12
12.5.4 The isr Threads 12–13

12.6 Synchronization 12–13
12.6.1 Locking 12–14
12.6.2 Socket Locks 12–14
12.6.3 Internet Domain Locks 12–16
12.6.4 UNIX IPC Socket Pairs 12–18
12.6.5 The Domain Funnel 12–18

12.7 Memory Management 12–21
12.7.1 Mbufs and Clusters 12–22
12.7.2 The mbuf Data Structure 12–23
12.7.3 Allocating mbufs 12–25
12.7.4 External Data 12–25

12.8 Sockets Security Extensions 12–26

Chapter 13. The OSF/1 STREAMS Framework 13–1

13.1 Overview 13–2

13.2 The STREAMS Programming Interface 13–4

13.3 STREAMS Operations 13–5
13.3.1 STREAMS as a Device Driver 13–5
13.3.2 Flow of Control Basics 13–6
13.3.3 Stream Head Routines 13–7
13.3.4 Operating System Requests 13–7

13.4 Scheduling and Flow Control 13–8

13.5 Synchronization 13–9
13.5.1 Synchronization Queue Structures 13–11
13.5.2 Changes to Standard STREAMS Structures 13–11

13.5.3 Executing the Synchronization Queue 13–13
13.5.4 Acquisition of Multiple Resources 13–16
13.5.5 Synchronization with Interrupts 13–16
13.5.6 Synchronization of sleep( ) Calls 13–18
13.5.7 Synchronization of timeout( ) and bufcall( ) 13–19

13.6 Memory Allocation 13–19
13.6.1 The bufcall( ) Routine 13–20
13.6.2 Interaction with mbufs 13–20

13.7 Cloning 13–21

13.8 Welding 13–22

13.9 Multiplexing 13–23
13.9.1 Multiplexing Lower Streams 13–24
13.9.2 Unlinking Multiplexed Lower Streams 13–25

13.10 Initialization and Configuration 13–26
13.10.1 Driver and Module Configuration Options 13–26
13.10.2 Synchronization Levels 13–27

13.11 Streams Security Extensions 13–29

Chapter 14. OSF/1 Logical Volume Manager 14–1

14.1 Overview 14–1

14.2 LVM Terms and Concepts 14–3
14.2.1 LVM Component Terms 14–4
14.2.2 Mirroring 14–6
14.2.3 Quorums 14–6
14.2.4 Logical-to-Physical Mapping 14–6

14.3 LVM Disk Layout 14–8
14.3.1 Physical Volume Reserved Area 14–8
14.3.2 Volume Group Reserved Area 14–9
14.3.3 User Data Area 14–11
14.3.4 Bad Sector Relocation Pool 14–11

14.4 Programming Interfaces 14–12
14.4.1 User Application Programming Interface 14–12
14.4.2 Administrative Application Programming Interface 14–12

14.5 LVM Device Driver Architecture 14–13
14.5.1 Data Structures 14–13
14.5.2 Driver Entry Points 14–15
14.5.3 Flow of Control 14–16

14.6 Driver Theory of Operation 14–17

14.7 LVM Configuration and I/O Layer 14–18
14.7.1 Driver Dynamic Configuration 14–18
14.7.2 Volume Group Configuration 14–18
14.7.3 Raw I/O Layer 14–21

14.8 Strategy Layer 14–21

14.9 Mirror Consistency Management Layer 14–22

14.10 Scheduler Layer 14–23
14.10.1 Scheduling Policies 14–23
14.10.2 Scheduler Operations 14–24

14.11 Status Area Manager 14–25

14.12 LVM Physical Layer 14–26
14.12.1 Revectoring Known Defects 14–26
14.12.2 Detecting New Defects 14–26
14.12.3 Relocating and Repairing Defects 14–27
14.12.4 Dynamic Detection, Relocation, and Repair 14–27

Chapter 15. Security 15–1

15.1 Security Overview 15–2

15.2 The Orange Book Model 15–4

15.3 Security Extensions 15–5

15.4 The Trusted Computing Base 15–7

15.5 Security Policy Architecture 15–11
15.5.1 Security Policy Modules 15–14
15.5.2 Security Policy Daemons 15–15
15.5.3 Security Policy Driver 15–16
15.5.4 Security Policy Database Manager 15–16
15.5.5 Interactions Example 15–17

15.6 Privileges and Authorizations 15–18

15.7 Security Administration 15–23

15.8 The Discretionary Access Control Policy 15–24
15.8.1 Discretionary Access Contol Components 15–24
15.8.2 Access Control Lists 15–26
15.8.3 Discretionary Access Control Privileges 15–28
15.8.4 ACL Representations 15–28
15.8.5 Example: Changing an ACL 15–29

15.9 Mandatory Access Control 15–30

15.9.1 Mandatory Access Control Components 15–31
15.9.2 MAC Privileges 15–32
15.9.3 MAC Access Decisions 15–34
15.9.4 MAC System Calls, Library Routines, and Commands 15–34
15.9.5 MAC Database Protection Principles 15–36

15.10 Authentication Subsystem and the Security Databases 15–37
15.10.1 Authentication Database 15–39

15.11 Audit Subsystem 15–44
15.11.1 Audit Subsystem Components 15–46
15.11.2 Audit Data Flow 15–49
15.11.3 Audit Record Formats 15–51
15.11.4 Audit Control Flow 15–53

15.12 File System Security Extensions 15–56
15.12.1 Mount Table Security Extensions 15–56
15.12.2 Vnode Security Extensions 15–57
15.12.3 Vnode Security Attributes 15–57
15.12.4 Vnode Security Routines 15–59
15.12.5 Superblock Modifications (UFS File System Type) 15–59
15.12.6 On-Disk Inode Extensions (UFS File System Type) 15–60
15.12.7 In-Core Inode Extensions (UFS File System Type) 15–60

15.13 STREAMS Security Extensions 15–61
15.13.1 Local Access Control 15–62
15.13.2 Internal Interfaces 15–63

15.14 Socket Security Extensions 15–63
15.14.1 Socket Data Structures 15–64
15.14.2 Socket Control Flow 15–64

15.15 Loader Security 15–65

15.16 Mach Subsystem Security 15–66

15.17 Modified Data Structures 15–67

15.18 New Data Structures 15–68

Glossary GL-1

Index Index-1

List of Figures

Figure 3–1. Tasks Sharing Access to Ports Using Private Port Rights . . . 3–5
Figure 3–2. Implementation of a Mach Virtual Address Space 3–10
Figure 3–3. A VM Object and Its Memory Object 3–12
Figure 3–4. The Mapping of Logical Page to Page Frames 3–14
Figure 3–5. Relationship Between an Address Map and Its Pmap 3–17
Figure 4–1. Structure of a Process in OSF/1 4–3
Figure 4–2. The Exception Handling Model 4–13
Figure 5–1. Suspend Mechanism State Diagram 5–12
Figure 5–2. Event-Wait Mechanism State Diagram 5–17
Figure 5–3. State Transition of a Thread in an Uninterruptible Sleep . . . 5–18
Figure 5–4. The Default Processor Set 5–20
Figure 5–5. An Application Allocates a Processor Set 5–21
Figure 5–6. The Application Requests Processors; the Kernel Assigns Processors 5–22
Figure 6–1. Implementation of Task Address Space 6–2
Figure 6–2. A vm_map Structure and Its vm_map_entry Structures 6–4
Figure 6–3. A vm_map_entry Structure and the VM Object It Maps . . . 6–6
Figure 6–4. Changing Protection on a Range of Virtual Memory 6–7
Figure 6–5. Implementation of a UNIX Process Address Space 6–9
Figure 6–6. Two Tasks Share Data Copy-on-Write 6–11
Figure 6–7. Task A Writes Data 6–12
Figure 6–8. Task B Writes Data 6–14
Figure 6–9. Tasks B and C Share Data Copy-on-Write 6–16
Figure 6–10. Task C Writes Data 6–17

Figure 6–11. Task B Writes Data, Creating a Shadow Tree 6–18
Figure 6–12. Pruning the Shadow Tree 6–20
Figure 6–13. Pruning the Tree Further 6–21
Figure 6–14. Tasks A and B Share Permanent Data Copy-on-Write 6–23
Figure 6–15. Task A Writes Data, Pushing a Page to the Copy Object . . . 6–24
Figure 6–16. Task B Writes Data 6–25
Figure 6–17. A Share Map 6–28
Figure 6–18. The Kernel's Address Map with Submaps 6–31
Figure 7–1. Page Clusters 7–5
Figure 7–2. The Target Page 7–8
Figure 7–3. Private Pages 7–12
Figure 7–4. Shared Memory Server Write Operation 7–18
Figure 7–5. Shared Memory Server Read Operation 7–19
Figure 8–1. The Loader Context 8–13
Figure 8–2. Known Modules List Example 1 8–15
Figure 8–3. Known Modules List Example 2 8–16
Figure 8–4. Known Modules List Example 3 8–16
Figure 8–5. Kernel Load Relocation 8–22
Figure 8–6. Layout of the Preload Cache Data File 8–23
Figure 9–1. Interrupt Handling 9–5
Figure 9–2. Device Driver Configured into Kernel Tables 9–7
Figure 10–1. Internationalization Subsystem Application Programming Interface 10–10
Figure 10–2. Internationalization Objects 10–11
Figure 10–3. Basic Stream for Terminal Devices 10–16
Figure 10–4. Basic Stream for Pseudoterminal Devices 10–18
Figure 11–1. Architecture of the File Management System 11–2
Figure 11–2. File Descriptor Reference to Open File Description 11–3
Figure 11–3. A Process and Its Open File Descriptions 11–5
Figure 11–4. Processes Sharing a Vnode 11–6
Figure 11–5. Example of OSF/1 VFS File Tree 11–9
Figure 11–6. Example of Data Structures for a Mounted File System 11–22

Figure 11–7. Device Special Files Data Structure 11–26
Figure 11–8. Buffer Cache and Vnode Data Structure Interaction 11–30
Figure 12–1. The domain Structure 12–4
Figure 12–2. The socket Data Structure 12–6
Figure 12–3. Managing Network Interrupts 12–11
Figure 12–4. Internet Domain Locking 12–16
Figure 12–5. The Domain Funnel 12–20
Figure 12–6. Components of the mbuf Data Structure 12–24
Figure 13–1. Flow in a Stream 13–4
Figure 13–2. An Example of Synchronization Queue Execution 13–15
Figure 13–3. Lower Streams Multiplexed to a Master Stream 13–24
Figure 14–1. Relationship of the LVM to Other System Components 14–3
Figure 14–2. A Mapping of Logical to Physical Volumes 14–7
Figure 14–3. Physical Volume Layout 14–8
Figure 14–4. Data Structures Describing a Volume Group 14–15
Figure 15–1. The OSF/1 Security Policy Architecture. 15–12

List of Tables

Table 10–1. ISO 8859 Code Sets 10–13
Table 10–2. OSF/1 Japanese EUC Code Set Encoding 10–14
Table 10–3. OSF/1 SJIS Encoding Method 10–15

Preface

The Open Software Foundation (OSF) was formed in May, 1988 specifically to develop software technologies and make them available on fair and reasonable terms. The Foundation's charter includes the following: to develop an open computing environment that employs a standard set of interfaces for programming, communications, networking and system management, in order that software applications may become uncoupled from specific hardware platforms.

OSF/1 is an advanced UNIX operating system developed to provide both application portability and powerful operating system functionality. Its first release was in December, 1990 and numerous updates have been developed since.

The *Design of the OSF/1 Operating System* describes the major features of the OSF/1 operating system and discusses the design issues involved in implementing these features.

Audience

The *Design of the OSF/1 Operating System* is addressed primarily to operating system developers and others who are interested in operating system internals. The discussion assumes that readers are familiar with operating system fundamentals and have a strong UNIX background. Chapter 2 provides some of this background for readers who require it.

Applicability

This is Version 1.0 of this document. It applies to Release 1.2 of the OSF/1 operating system.

Document Usage

The book's chapters are organized into three parts: Chapters 1 through 3 provide overview and introductory material, chapters 4 through 7 describe the core kernel portion of the system, and chapters 8 through 15 describe the system services:

- Chapter 1 provides an overview of OSF/1.
- Chapter 2 provides an overview of UNIX processes and the services the kernel provides to processes.
- Chapter 3 provides an overview of the Mach technology that is the basis of OSF/1's core services.
- Chapter 4 describes the structure and management of processes in OSF/1.
- Chapter 5 describes OSF/1's scheduling subsystem.
- Chapter 6 describes the address space implementation of the OSF/1 virtual memory subsystem.
- Chapter 7 describes the memory management portion of the OSF/1 virtual memory subsystem.

- Chapter 8 describes the OSF/1 program loader.
- Chapter 9 describes how OSF/1 supports dynamic loading and configuration of kernel subsystems.
- Chapter 10 describes the OSF/1's internationalization subsystem and includes a discussion of how the kernel's STREAMS-based tty subsystem supports internationalized applications.
- Chapter 11 describes how OSF/1 manages files.
- Chapter 12 describes the implementation of the OSF/1 sockets framework.
- Chapter 13 describes the implementaion of the OSF/1 STREAMS framework.
- Chapter 14 describes the Logical Volume Manager, OSF/1's disk storage management system.
- Chapter 15 describes the security features of OSF/1.

Related Documents

The following OSF/1 documents are currently available from Prentice Hall:

- *Design of the OSF/1 Operating System*
- *OSF/1 User's Guide*
- *OSF/1 Command Reference*
- *OSF/1 Programmer's Reference*
- *OSF/1 System and Network Administrator's Reference*
- *OSF/1 Network Applications Programmer's Guide*
- *Application Environment Specification (AES) Operating System Programming Interfaces Volume*

In addition, versions of the following documents may be available from your system vendor:

- *OSF/1 System Programmer's Reference Volume 1*
- *OSF/1 System Administrator's Guide*
- *OSF/1 Network and Communications Administrator's Guide*
- *OSF/1 System Porting Guide*
- *OSF/1 System Extension Guide*
- *OSF/1 Security Features User's Guide*
- *OSF/1 Security Features Programmer's Guide*
- *OSF/1 Security Features Administrator's Guide*
- *OSF/1 Security Detailed Design Specification*
- *OSF/1 POSIX Conformance Document*

Problem Reporting

If you have any problems with the software or documentation, please contact your software vendor's customer service department.

Acknowledgements

This book is the result of the work of a dedicated group of participants on the OSF/1 technology team. The following writers, editors, engineers, and managers were directly involved in the creation of this work:

Bill Bryant, Noreen Casey, Josh Goldman, Bernice Moy, Peter Neilson, Tom Talpey, Willie Williams, Jeff Carter, Maureen Ellenberger, Susan Kegel, George Feinberg, Tom Doeppner, David Black, Al Lehotsky, and Jeff Collins.

This book also reflects the work of many people who all made important contributions to the OSF/1 technology. This book would not be possible

without the coordinated efforts of the following people from the entire OSF/1 team:

Francesco Aliverti-Piuri, Larry Allen, David Anastasio, Matthias Autrata, Randy Barbano, Bruce Bauman, Bob Binstock, Peter Bishop, Don Bolinger, John Bowe, Cathleen Brecht, John Brezak, Mark Brown, Julie Buckler, Tim Burchell, Lorraine Burrage, Yakov Burtov, Bob Canavello, Frank Casper, David Chinn, Dan Christians, Mike Collison, Elizabeth Connolly, Robert Coren, Darrrell Crow, Beth Cyr, Fred Dalrymple, Greg Depp, Marcia Desmond, Robert DiCamillo, Chris Doherty, Nick Dokos, Michelle Dominijanni, Ed Doyle, John Dugas, Eric Dumas, Kathy Duthie, Jack Dwyer, Gary Fernandez, Tony Fiore, Ed Frankenberry, David Gillespie, Frank Ginac, Martin Gosejacob, Roger Gourd, Marilyn Grady, Steve Grainger, Courtney Grey, Kathy Grimaldi, Paul Groff, Michael Gross, William Hankard, Scott Hankin, Peter Harbo, Bob Hathaway, Martha Hester, Jeanette Horan, Doug Hosking, Jean Hsiao, Thomas Jordahl, Stephen Kafka, Larry Kaplan, Paul Karger, Jerry Kazin, Dave Kirschen, Ned Kittlitz, Roy Klein, Steve Knight, Natalia Kogan, Cheryl Korizis, Sharon Krause, Andreas Kroneberg, Salvatore LaPietra, Alan Langerman, Chain Lee, Lenny Lefort, Steve Lewontin, Hal Lichtin, Chi Hwei Lin, Sally Long, Sue LoVerso, Marty Lynch, Rod MacDonald, Joe Maloney, Steve Marcie, Glenn Marcy, Andy Maretz, Mark Marino, Norbert Marrek, Sandra Martin, Ray Mazzaferro, Andy McKeen, Cindy McKeen, Michael Meissner, Jody Menton, Franco Miralles, Dave Mitchell, Mariko Mori, John S. Morris, Richard Morris, Linda Mui, Betty Newman, Ralf Nolting, Rose O'Donnell, Jay Orsini, Charles Pacheco, Noemi Paciorek, Maryanne Paratore, Simon Patience, Ellen Patton, Per Pedersen, Grace Perez, Staffan Persson, Jacqueline Philbin, James Pitcairn-Hill, Damon Poole, Dan Powers, Paul Rabin, Vella Raman, Ron Rebeiro, Renee Rice, Uwe Richter, Jack Rieden, Philip Rockwood, David Rodal, John Rousseau, Ken Sallale, Arno Schmidt, Ken Seiden, Peter Shaw, Eric Shienbrood, Harminder Singh, Bruce Smith, Jennifer Steiner, Kevin Sullivan, Susan Teto, Peter Thomas, Kevin Till, James Van Sciver, Kevin Wallace, Susanna Wallace, Ping Wang, Peter Watkins, Melanie Weaver, Doug Weir, Jeff Whalen, Jie Yao, and Glenn Zazulia.

Chapter 1

The OSF/1 Operating System

OSF/1 is an advanced UNIX operating system. It provides an applications programming environment that furnishes, in a single environment, many of the features found in different UNIX programming environments. The OSF/1 kernel provides powerful operating system functionality that can be used to implement features not generally associated with traditional UNIX systems. The OSF/1 programming environment and the powerful facilities of the kernel implement an advanced software environment that supports applications portability and establishes a basis for the development of future operating systems.

OSF/1 is an *open system*; its specification conforms to public, international standards and it is widely compatible with systems from a variety of manufacturers. It is easy to port and can be configured to run on machine architectures ranging from personal computers to high-performance workstations and multiuser timesharing machines. It supports symmetric multiprocessing and distributed computing environments, and is designed to be easily extensible.

1.1 UNIX Functionality

From an application programmer's point of view, OSF/1 provides a UNIX programming environment. In OSF/1, programs are executed as *processes*, and the system provides all process-related facilities generally associated with UNIX systems. OSF/1 is compatible with software developed both for Berkeley 4.3 and 4.4 as well as System V releases 3 and 4. The operating system supports the following standards and specifications:

- ISO/IEC 9945-1:1990 (POSIX.1). In those instances where this standard indicates alternatives in functionality, OSF/1 uses the functionality specified by the Federal Information Processing Standard (FIPS) 151-1.
- ISO/IEC 9945-2:1992 (POSIX.2).
- *X/Open Portability Guide, Issue 4* (XPG 4).
- *System V Interface Definition, Issue 3* (SVID 3).
- OSF's *Application Environment Specification (AES) Operating Systems Programming Interfaces*.
- Berkeley 4.3 and 4.4 application interfaces.
- ISO/IEC 9899:1990 (C Programming Language).

OSF/1 also provides the BSD UNIX File System (UFS), the System V S5 File System, and an unencumbered implementation of the Sun Network File System (NFS).

OSF/1 provides well-known UNIX interprocess communication mechanisms, including BSD sockets and the X/OPEN Transport Interface (XTI). The Internet protocol family is provided under both interfaces, providing the familiar IP, TCP, and UDP protocols. The system also provides a STREAMS framework, which can be used to implement device drivers and network protocols in a modular fashion. In OSF/1, STREAMS is used to implement the terminal subsystem.

1.2 Advanced Features

In addition to providing functionality associated with traditional UNIX systems, OSF/1 provides many features that augment UNIX functionality. These features include the following:

- Efficient operation in uniprocessor and multiprocessor environments.
- Support of multithreaded applications; that is, applications that contain multiple threads of control. In a multiprocessor environment, the threads of a multithreaded application can execute in parallel. (The POSIX P1003.4 draft 6 programming interface is provided for threads.)
- Application access to the powerful virtual memory and messaging primitives of the core kernel.
- Support of shared libraries. Processes can share a single copy of system libraries. This greatly reduces the size of a program's executable file.
- A flexible user space program loader that supports different object file formats, shared libraries, and dynamic loading and unloading.
- Dynamic loading and unloading of many kernel modules. This feature allows system administrators to configure the kernel at runtime.
- A Logical Volume Manager that allows file systems to span physical devices, and allows such volumes to enhance data availability and reliability.
- An object-oriented internationalization subsystem that allows applications to operate using the language, codeset, and cultural conventions appropriate to the user's environment.
- A security subsystem that supports both B1 and C2 security classes, as defined by the U.S. government's National Computer Security Center.

1.3 What is OSF/1?

OSF/1 is an integration of operating system and application programming interface (API) technologies. This book focuses primarily on the operating system, or kernel, portions of OSF/1.

The OSF/1 kernel consists of two logical elements: the *core kernel* and the *system services*.

The core kernel provides the basic hardware support and the kernel's memory management and scheduler subsystems. It is derived from the Mach operating system, which was developed at Carnegie-Mellon University. Currently, the OSF/1 core kernel is based on Mach 2.5. Mach provides a small set of operating system objects and operations on those objects. These objects and operations can be used to implement different operating system personalities. OSF/1 uses them to implement a UNIX personality.

The system services provide the operating system facilities that are used directly by applications programs, and provides the services generally associated with UNIX environments.

1.3.1 Tasks and Threads

The OSF/1 operating system abstracts a process's components into a *task*, which represents a set of system resources including an address space, and a *thread*, which represents the process's thread of control.

OSF/1 uses the task and thread objects to implement processes. A standard process consists of a task with a single thread; however, OSF/1 also supports processes that contain multiple threads. Interfaces are provided that allow processes to create and control such threads.

The thread construct is enhanced by a powerful and flexible scheduler provided by the core kernel. The scheduler provides policies and extensibility which can be used to support UNIX as well as non-timesharing models for other operating system environments.

1.3.2 Virtual Memory and Memory Management

OSF/1 incorporates an innovative memory management system that is highly portable. The implementation of virtual memory is cleanly divided into machine-independent and machine-dependent pieces; all machine-dependent operations are implemented in a single module called the **pmap** (physical map) module. This module manages the data structures and the hardware's *memory management unit* (MMU) to perform address translation.

All virtual memory state is managed with machine-independent data structures; the system uses the machine-dependent data structures to cache address translations only as they are needed. The **pmap** module performs these machine-dependent manipulations.

The virtual memory system provides functionality not generally associated with traditional UNIX systems. For example, the external memory management interface supports the development of user space memory managers which can be used to allow applications to map application-specific objects into their address spaces.

The memory management system makes extensive use of *copy-on-write* techniques to copy memory between processes. These techniques are used to optimize virtual memory operations that have traditionally been quite expensive, such as the **fork( )** system call.

The memory management system also supports large sparsely filled virtual address spaces, which allows OSF/1 to provide support for shared libraries.

Virtual memory and memory management are discussed in Chapters 6 and 7.

1.3.3 File Management

OSF/1 file management consists of three distinct subcomponents:

- The per-process file tables implement traditional UNIX file descriptors, with the important addition of support for fully threaded applications.

- The VFS provides a single interface to the file systems so that the system can perform operations on a file in the same manner. The file systems, in turn, are implemented and interface to the VFS in a consistent manner.
- OSF/1 provides three file systems, and can be extended to provide others:
  - — The UNIX File System (UFS)—a parallelized implementation of the Berkeley Fast Filesystem.
  - — A version of the System V File System for compatibility.
  - — A Network File System compatible with the Sun Microsystem NFS.

File management and file systems are discussed in Chapter 11.

1.3.4 Networking

OSF/1 provides three facilities that serve as frameworks for networking:

STREAMS — A kernel facility that provides a communications path between a user process and various classes of device drivers.

Sockets — A Berkeley UNIX kernel facility that provides communications for user processes to networks. Sockets also provide a specialized application programming interface.

The X/OPEN Transport Interface (XTI) — An applications programming interface that provides communications between user processes and kernel-provided transport layers, which in turn access network devices.

OSF/1 provides communications through the support of the Internet protocol suite, which consists of a number of protocols, including the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP).

1.3.5 STREAMS

The STREAMS framework provides a way to implement communications software, such as network protocols or terminal protocols. A communications path, called a stream, provides the interface between the user process and a driver.

STREAMS also provides a facility for incorporating character I/O device drivers into the kernel. It includes a consistent set of user and kernel space interfaces that provide a standard interface for devices to communicate with the operating system. It provides the implementation framework for terminals, communications services and protocols.

STREAMS is discussed in Chapter 13.

1.3.6 Sockets

OSF/1 sockets is an implementation of the Berkeley 4.3 and 4.4 sockets technology. Applications that use these sockets versions will also operate under OSF/1.

The sockets framework has a user level and a kernel level. At the user level, the sockets framework supports system calls that access sockets, and at the kernel level, the sockets framework provides the underlying support for the Internet suite of protocols. The OSF/1 kernel-level sockets framework allows protocols and families of protocols to be dynamically configured into the system.

The sockets framework is fully parallelized; therefore, it can work in both uniprocessor and multiprocessor environments. The OSF/1 sockets framework can also work with protocols whose code has or has not been parallelized.

Sockets are discussed in Chapter 12.

1.3.7 XTI

OSF/1 XTI is a STREAMS-based implementation of the X/OPEN XTI programming interface. XTI is an enhancement of the AT&T TLI (Transport Layer Interface). It provides an interface to the transport layer of diverse protocols conforming to the seven-layer International Standards Organization Open Systems Interconnection (ISO OSI) model. The transport layer's job is to provide end-to-end communications between processes on different machines across a network.

The XTI is independent of the transport protocols used by a system and independent of the machines. The XTI allows applications to create connections to remote peers, to transfer data, and to terminate connections through a series of interfaces. The OSF/1 XTI is fully parallelized, and it can operate in both uniprocessor and multiprocessor environments.

1.3.8 Internationalization

In the past, UNIX systems have supported only English-speaking environments. However, in recent years, UNIX systems have experienced more international use. The entire OSF/1 system has been internationalized according to specifications set down by standards bodies and user groups such as POSIX and X/OPEN, and OSF/1 includes an internationalization subsystem that supports internationalized applications

OSF/1 implements internationalization support through a set of *locales*. Each locale specifies a software environment that supports the language and customs associated with a particular geographic region.

An application determines the current locale at runtime, usually by means of the user's environment variables. The application then calls the internationalization subsystem to load the tables and algorithms that implement the locale. When the application performs an operation that is locale-dependent, the routine that performs the operation uses the algorithm and data specific to the locale.

The internationalization subsystem is discussed in Chapter 10.

1.3.9 Terminals

The OSF/1 terminal subsystem (tty/pty) provides traditional UNIX tty functionality, and is compliant with POSIX.1. The important aspect of the OSF/1 terminal subsystem is internationalization.

All UNIX terminal subsystems use elements called line disciplines to perform terminal input and output processing. A line discipline is one of the software components that a tty/pty driver employs to process input characters.

In OSF/1, the terminal subsystem is a STREAMS-based implementation. This is key to a more modular approach to internationalizing the terminal subsystem. In this approach, the main component of the terminal subsystem is a line discipline that provides the traditional capabilities. To accommodate the needs of non-English locales, the line discipline is augmented by converter modules that provide communication in the character set of the locale.

The terminal subsystem is described in Chapter 10.

1.3.10 Logical Volume Manager

The OSF/1 Logical Volume Manager (LVM) extends standard disk management capabilities. A UNIX disk driver stores and retrieves data to and from a single physical disk unit. The LVM interfaces associate one or more disk drivers into a single logical disk, enabling the system administrator to enlarge the virtual storage space, span the data storage across disks, and replicate data (known as mirroring).

The LVM subsystem consists of the following components:

- Logical and physical volumes, where logical volumes represent virtual disks and physical volumes represent physical storage, such as a disk or disk partition.
- A logical volume device driver, which interacts with the actual disk driver(s) to manipulate data. The application sees the logical volume device driver as a single disk driver.

- A programming interface, through which a user can manage the volumes.

The LVM is discussed in Chapter 14.

1.3.11 Program Loader

Program loaders load executable object files into memory and prepare them for execution. They often work with linkers to resolve external symbol references and perform relocation before programs can be executed with the **exec( )** function. In traditional UNIX systems, the program loader function is closely tied with the linker, and is commonly known as the linker/loader (**ld**).

In OSF/1, the program loader is separate from the linker. When the program is linked, a binary file is created, but not all external or symbol references may be resolved. At the time the **exec( )** function is called to actually execute the program, the kernel invokes the OSF/1 program loader to resolve remaining symbol references and to load the file for execution.

The OSF/1 program loader extends the functionality generally associated with traditional UNIX program loading. In addition to resolving and relocating symbols, it supports multiple object file formats, shared libraries, and runtime dynamic loading and unloading. Except for the **exec( )** function, the loader is implemented entirely in the user space.

The OSF/1 program loader supports the implementation of shared libraries. In traditional UNIX linker/loaders, when a symbol is resolved and its definition is found in a library, the linker/loader copies the module into the program's executable image. This method can be inefficient when programs are large, or when a large number of references are resolved. Shared libraries overcome these problems by providing a single copy of each of their routines to be shared by many processes running on the system.

The OSF/1 program loader provides a *package* abstraction to help in symbol resolution with shared libraries. The program loader uses each package to map symbol names to the appropriate library without having to stamp a pathname in a binary image. This allows maximum flexibility and mobility.

The same program loader also provides a kernel loading capability, which enables the kernel to dynamically load and unload modules. This allows the kernel to add and remove, at runtime, new system services, file systems, device drivers, network protocols, and streams modules.

The program loader is discussed in Chapter 8.

1.3.12 Security

OSF/1 provides a security subsystem that complies with certain elements of the U.S. government's *U.S. Department of Defense Trusted Computer System Evaluation Criteria* (TCSEC, or the Orange Book). This is the definitive guide to the development and evaluation of trusted computer systems. The security subsystem enables OSF/1 to be configured for varying levels of security, including both the basic features and supersets of those required for C2 and B1 level security.

The security subsystem can be viewed as both code and process. The code part of the security subsystem consists of functions and kernel compilation conditionals that enable the different levels of security. The process part of the security subsystem requires that each application use the security functions and run on a secure kernel for the existing security features to take effect.

Security extensions have been added to many elements of the OSF/1 operating system, including the kernel itself, kernel services such as the file systems, the programming interface, and user-level commands.

Security is discussed in Chapter 15.

1.3.13 Scalability and Dynamic Configuration

Traditional UNIX systems are limited in their capabilities to be reconfigured or scaled up or down easily during runtime. For example, reconfiguring a traditional UNIX system to add or remove a file system from the kernel would require making the configuration changes, rebuilding the kernel, and restarting the operating system.

The OSF/1 kernel can be dynamically tuned and reconfigured while the system is running. The following subsystem components allow dynamic configuration:

- The filesystem framework allows filesystems to be dynamically added and removed.
- The STREAMS framework allows STREAMS-based drivers and modules to be dynamically added and removed.
- The sockets framework allows families of protocols to be dynamically added and removed.
- The terminal subsystem, which is STREAMS-based, allows STREAMS-based line disciplines and drivers to be dynamically added and removed, and also to be configured onto specific terminals and ports.
- The Logical Volume Manager allows the LVM device driver to be dynamically added and removed, and to dynamically configure logical volumes.
- The system call framework allows the dynamic addition of new system services.
- Almost all device drivers can be dynamically added and removed.

One of the advantages of a scalable OSF/1 system is that it can provide a version of the OSF/1 system with a minimum of its possible subsystems actually configured. Because of the dynamic capabilities, such a system can be easily expanded as the needs require without causing system downtime.

Configuration of the kernel is discussed in Chapter 9.

1.4 The Future of the OSF/1 Design

Currently, OSF/1 integrates the core kernel services with the system services into one monolithic kernel. In future versions, the kernel will contain only the primitive objects and operations provided by a *microkernel*; most of what is now contained in the system services will be moved into its own, separate address space.

The microkernel configuration has many advantages over the traditional monolithic kernel. For example, a single machine running a single core kernel may simultaneously run multiple system services, such as multiple operating systems "personalities," much as multiple user tasks are run today. Additionally, the system services could run in a distributed fashion across a network of machines running a common core kernel. Such "massively parallel," "cluster," or "multicomputing" machines provide an entirely new dimension to computing.

Chapter 2

Overview of UNIX Processes and the UNIX Kernel

A UNIX kernel is responsible for managing, on behalf of user applications, the system's resources. These resources include the CPU, resident memory, and all peripheral devices that are configured into the system, including any disk drives, tape drives, terminals, printers, and network hardware.

The kernel uses *processes* to manage the execution of applications. The process construct allows the kernel to control the use of system resources so that

- All currently active applications have reasonable access to system resources.
- Applications cannot inadvertently or deliberately interfere with one another's access to the resources.

This chapter provides an overview of the UNIX process construct and the services the kernel provides to processes. This chapter also describes the major operations performed by the kernel as it manages the system's resources. If you are familiar with UNIX operating system internals, you may want to skim this chapter, or skip it altogether.

2.1 Process Address Spaces

When a progam is compiled, the compiler creates the program's executable file, also referred to as the *executable image*. The kernel uses this file to create a *logical address space* that contains the following sections of data:

- The program's *text* section, which contains the executable instructions.
- The program's *initialized data*. This data is *global data,* which will be accessible to the program's main routine and all of the subroutines defined in the program and in any libraries that the program references.
- The program's *uninitialized data*. The compiler allocates storage for this data, but the data is not initialized until runtime. This data is also global data.

The executable file also includes a header, which specifies the location and size of each of the data sections. When the program is being prepared for execution, the system's program loader uses the header information to set up the process's virtual address space.

When the process's address space is set up, it contains a text section, an initialized data section, an uninitialized data section, and two additional sections: the process's *heap,* and the process's *user stack.* The heap contains memory that the process explicitly acquires during its execution. Typically, a process uses heap memory to store dynamically required data structures. When a new data structure is required, the process executes a call to **malloc( )** to allocate the memory. When the data structure is no longer needed, the process can execute a call to **free( )** to free the memory. Like the initialized and uninitialized data, the heap data is global.

In contrast to the data contained in the heap, the data contained on the stack is local data, which is accessible only to the process's currently active routine. A process's user stack grows and shrinks dynamically as needed.

2.2 Process Management System Calls

The UNIX kernel provides a set of process management system calls that allow processes to create other processes, to manage the execution of related processes, and to terminate themselves or the processes they control. These include **fork( )**, **exec( )**, **wait( )**, and **exit( )**.

Processes use the **fork( )** and **exec( )** system calls to create processes and execute new programs, respectively. The **fork( )** system call creates a new process by duplicating the address space of the calling process. The calling process is referred to as the *parent* process and the new process is referred to as the *child* process. Upon successful completion of **fork( )**, the parent and child have duplicate address spaces and are executing the same program.

The **exec( )** system call allows a process to execute a new program by loading the program into the process's address space. (Actually, **exec( )** is a family of system calls, but it is referred to as a single system call to simplify the discussion.) Generally, a child process that is to execute a new program issues a call to **exec( )** after the call to **fork( )**.

A parent process may choose to wait for its child to complete execution before resuming execution itself. For example, the shell does this when executing commands in the foreground. The user enters a command to the shell, the shell uses **fork( )** to create a new process, the new process calls **exec( )** to load the command's program, and the shell waits for the program to complete execution.

A process that needs to wait in this fashion does so using the **wait( )** system call. This system call suspends the calling process's execution until the child process either terminates or suspends itself. It is called with a *status* argument that the system uses to inform the waiting process about the exit or suspend status of the child process. When the child exits or suspends itself, the system copies its status to the *status* variable and allows the parent process to resume execution. The parent can examine the *status* variable to determine what happened to the child.

When a process wants to explicitly terminate its execution, it does so using the **exit( )** system call. This system call releases all of the process's system resources and may send a *signal* to the process's parent process to indicate that the child has exited. The signal subsystem and a process's state with respect to signals are discussed in Chapter 4.

2.3 Process States

Each process has a set of *states* with respect to the system. These include the following:

Execution State

A process's execution state specifies whether or not the process is executing or executable. With respect to this state a process is either:

- Executing, or executable and waiting to be scheduled.
- Blocked while waiting for a system resource to become available. A process in this state is said to be *sleeping*.
- Suspended; that is, not executable, and not waiting for access to a system resource. For example, UNIX systems that support job control allow users to suspend the execution of a process from the terminal by entering the SUSPEND character (usually **Ctrl-Z**).

Scheduling State

A process's scheduling state indicates when the process will next be scheduled for execution. When a process is created, the kernel assigns it a scheduling priority. The kernel schedules the CPU by choosing the currently active process that has the highest priority.

Generally, a process that is executing, or has just executed, has a lower priority than a process that has not executed as recently. The kernel's scheduler subsystem periodically adjusts each process's scheduling priority so that all processes get equitable access to the CPU. Chapter 5 describes the OSF/1 scheduling subsystem.

File Descriptor State

The kernel maintains for each process a table of *file descriptors*, each of which represents a file or network connection that the process has access to. A process usually has at least three descriptors in its table; standard output is represented by descriptor 0, standard input by descriptor 1, and standard error by descriptor 2.

1 2

A process may inherit other file descriptors from its parent process when it is created. When the process opens a new file or network connection, the kernel places a new descriptor in the table. Each descriptor is associated with an underlying data structure that the kernel uses to manage the file or the network channel.

Process Identification and Relation States

Each process has a *user ID* that identifies the user who is responsible for the process, and a set of *group ID*s that identify what *user group*s the process's user belongs to. The kernel uses these IDs when determining whether or not to grant a process access to specific system resources such as files.

When the kernel creates a process, it assigns the process a unique process ID number (PID). Other processes may reference the process by its PID. A process also has access to the PIDs of any processes it has created as well as the PID of its parent process.

Traditionally, a process's states are maintained in its **proc** and **user** data structures. Chapter 4 describes the data structures used in OSF/1 to maintain process states.

2.4 Memory Management

All executing processes require access to resident memory and to the CPU; before a program can be executed, its instructions and data, or some portion of them, must be copied from the program's executable file in secondary storage into the hardware's resident memory. When the kernel schedules the process for execution, the kernel initializes the CPU's registers so that the CPU can locate the program's instructions and data. The CPU then executes the process.

2.4.1 Memory Management Techniques

In OSF/1, as in most recent UNIX implementations, processes do not need to be fully loaded in resident memory to execute. OSF/1 implements a form of memory management known as *demand paging*. The kernel loads a process's instructions and data only when the process needs them. Instructions and data that are not currently needed may reside in secondary storage until they are needed.

In early UNIX versions, processes had to be entirely resident in order to execute, and the operating system implemented a memory management policy that involved *swapping* processes in their entirety between secondary storage and resident memory. The computers that ran early UNIX systems had, by today's standards, very small resident memories. Because processes had to be fully resident to execute, a process could not be larger than the available physical memory. UNIX systems that support demand paging do not limit a process's size to the size of physical memory because a process does not need to be fully resident to execute.

In OSF/1, resident memory is conceptualized as a linear continuum of physical address space that is divided into fixed-length units known as *page frames*. Resident memory can be thought of as an array of physical page frames, with each page frame having a page frame number. Any location in physical memory can be specified by a page frame number and an offset into the page frame.

When the OSF/1 kernel initializes a process for execution, it does not load the entire process into resident memory. Instead, the kernel allows the process to begin executing and *pages in* the process's instructions and data *on demand*. When the CPU needs to access a particular instruction or data location that is not in resident memory, the kernel copies the appropriate page from secondary storage to one of the page frames in resident memory.

In a demand paging system, a process's pages of instructions and data are usually scattered throughout physical memory; they are rarely, if ever, placed contiguously. When many processes are active, resident memory contains some number of pages for each process, with the pages scattered throughout the array of page frames in an arbitrary fashion.

Managing the physical locations of a process's pages is the responsibility of the kernel, not the process. Like other UNIX systems, the OSF/1 kernel uses *memory mapping* techniques to present processes with simple address spaces. In memory mapping, a process's address space contains *logical*

addresses, which do not correspond directly to physical locations in resident memory. The kernel, with support from the hardware, maps these logical addresses to physical locations by means of a *memory map*.

In order to support memory mapped address spaces, the hardware must include a *memory management unit* (MMU), which uses a process's memory map to translate logical address references to their physical counterparts. In a memory mapped system, the CPU references instructions and data by presenting the MMU with logical addresses. The MMU translates each reference to a physical address, accesses the data, and presents it to the CPU.

The kernel implements separate address spaces by providing each process with its own memory map. When the kernel schedules a process for execution, it loads the MMU's registers so that the MMU can find the process's memory map. This operation is strictly controlled by the kernel so that the kernel can prevent user-level processes from accessing memory maps that are not their own. The kernel also prevents processes from arbitrarily changing their memory maps to map to other processes' data.

2.4.2 The Transparency of Memory Management

The kernel performs memory management operations transparently; user processes execute without being aware that their data and instructions are being paged in from secondary storage. When the CPU references a virtual address that is not mapped to an address in resident memory, the reference generates a *page fault exception*, which forces the CPU to stop executing the user process and execute the kernel's *page fault handler*. The page fault handler allocates a new page frame, pages in the required data, and updates the process's memory map so that when the process references addresses corresponding to the page, the MMU will translate the references correctly.

When the page fault handler has completed its operations, the kernel returns control to the user process. The CPU re-executes the instruction that generated the page fault and the process resumes execution without knowing that the pagein operation took place.

An executing process may page in a large number of pages during its lifetime. As a process continues to execute, it may no longer need some of the pages it has paged in. If a process is active for a long time, the kernel may need to reclaim some of the process's page frames so that they can be

allocated to other processes. The kernel has a *pageout daemon* that reclaims page frames that are allocated to processes but are not being referenced any longer.

The pageout daemon is a kernel process that executes when the number of unallocated page frames drops below a certain level. The pageout daemon finds page frames that have been allocated to processes but are not being actively used.

If a given page frame has data on it that needs to be saved, the pageout daemon arranges for the data to be written to secondary storage; when the data has been written, the pageout daemon reclaims the page frame. If a page frame contains data that has not been modified since it was paged in, the daemon reclaims the page frame immediately. When the pageout daemon reclaims a page frame from a process, it updates the process's memory map so that it no longer refers to the page frame.

Like the page fault handler, the pageout daemon performs its operations transparently; processes are unaware of its existence. If a process references instructions or data that are no longer in resident memory, the reference generates a page fault exception and the page fault handler brings the page back in.

2.5 Process Context and Context-Switching

A CPU always executes instructions *within the context* of the current process. In general, a process's context is specified by its memory map and by its *computational state*.

A process's computational state is specified by the contents of the CPU's registers as the CPU executes the process. The detailed characteristics of a CPU's registers are hardware-specific, but in general, CPUs include the following types of registers:

Program Counter

This register is the means by which the CPU finds the next instruction to execute. A CPU's behavior with respect to this register is hardware-dependent, but many CPUs increment this register at the time they are loading the current instruction so

that when the current instruction has been executed, the CPU can find and load the next instruction.

Stack Management Registers

The CPU uses these registers to locate and manipulate the process's stacks. In UNIX systems, a user process has two stacks: a user stack and a kernel stack. When a process executes in user mode, variables are stored on the user stack. When the process executes a system call, the system call's variables are stored on the kernel stack. Stack management is highly machine-dependent. The CPU must be able to determine which stack is currently active, and it must be able to locate variables on the stacks.

General Registers

These registers are used to store variables that the CPU needs to access quickly. Usually, the general registers hold operands that are being manipulated by the process's current state of execution. For example, if a process is executing a **for** loop that increments and tests a variable before looping, that variable is probably being stored in a general register.

A process's computational state is highly dynamic. The program counter changes with each instruction, and the stack management registers change each time the process executes a system call or subroutine.

Any of a number of events can interrupt a process's execution. When an interruption occurs, the kernel must save the process's computational state so that when the process resumes execution, it executes from the point of interruption.

When the kernel schedules a new process for execution, it switches the CPU's context from the previous process to the new process. The kernel saves the first process's register state in memory, purges the CPU's registers and MMU, and then loads the new process's register state into the CPU.

2.6 The UNIX Kernel and Its Services

It is common in discussions about UNIX to refer to the kernel as a separate entity that performs its operations independent of all user processes. For example, in a discussion about the memory management subsystem, it is often said that the kernel *pages in data from the disk* after *allocating resident memory* for the incoming data, and so on, as though the kernel performed these operations as a separate process or set of processes. Discussing the kernel using this convention is somewhat misleading because it implies that the kernel actively performs operations on its own initiative.

In fact, the kernel is essentially passive; the execution of kernel code is driven by events that are external to the kernel. These events can be classified as follows:

- A user process requests access to a system resource.
- A peripheral device is ready to perform an I/O operation.
- The hardware's clock interrupts the CPU's current state of execution, thereby causing the causing the kernel's *clock interrupt handler* to perform system management operations.

2.6.1 System Calls

A user process makes requests to the kernel via the *system call interface*. The system call interface is the set of routines that processes can use to access and manipulate system resources. For example, a process can access a given file by issuing an **open()** system call, and can read data from and write data to the file via the **read()** and **write()** system calls. The system call interface is a user process's only means for explicitly accessing system resources.

When a process executes a system call, it changes its *execution mode* from *user mode* to *kernel mode*. A process in user mode has access only to its instructions and data; it cannot access kernel instructions and data. A process in kernel mode can access kernel instructions and data. When a process enters kernel mode via a system call, it executes the kernel code that implements the system call. In other words, it has *entered the kernel*.

The kernel does not execute the system call's code; the process executes the code in kernel mode.

2.6.2 Program Exceptions

A program exception is an event caused by the currently executing process. Two types of program exceptions, page fault exceptions and exceptions caused by the issuing of system calls, have already been discussed.

A process may also generate exceptions that indicate programming errors. A process will generate an exception if it does any of the following:

- Divides by zero
- References an invalid address
- Executes a system call that does not exist

When a process generates a programming error exception, the CPU executes the kernel's trap handler. The trap handler diagnoses the problem and posts a *signal*, or software interrupt, to the process. The signal may force the process to terminate.

2.6.3 Peripheral Device Activity

Many system calls are requests for access to system resources associated with peripheral devices. For example, the **open()**, **read()**, and **write()** system calls are often requests for access to files on disk.

Each peripheral device that is configured into the system has a *device driver* that provides an interface between the kernel and the peripheral device. A typical UNIX kernel has separate device drivers for the system's disk and tape drives, terminals and printers, and network hardware such as Ethernet cards.

When a process executes a system call that requests access to a resource managed by a peripheral device, the kernel code executing the request accesses the system resource by invoking the appropriate device driver

routine. For example, when a process calls **open()** to get access to a file on disk, the kernel invokes code within the disk drive's device driver to enable the disk for operation.

Resources that are managed by peripheral devices are often not instantaneously available. For example, before a disk drive's controller can read data from the disk, it must wait for the data to spin under the disk drive's read head. The kernel cannot reliably predict when the disk drive controller will supply the requested data, so typically, when a process requests an **open()**, **read()**, or **write()**, the kernel code puts the process to sleep so that other processes can be executed while the first process waits for the resource to become available.

A peripheral device that is ready to furnish a resource must be able to inform the system that the resource is available. This is done through the *device interrupt* mechanism. For example, when a disk drive controller is ready to provide requested data, it posts an interrupt to the CPU. This interrupt forces the CPU to stop executing its current set of instructions and execute the disk drive's *interrupt handler* instead.

An interrupt handler is that part of a peripheral's device driver that manages the transfer of data between the kernel and the peripheral device. In the example, the disk drive's interrupt handler transfers the data from the disk drive to the kernel and wakes up the process that is waiting for the data. When the interrupt handler completes its execution, the CPU resumes executing the code it was executing before the interruption.

When the CPU executes interrupt handler code, it is in a mode of execution that is called an *interrupt level*.

2.6.4 The Hardware Clock

The hardware's clock interrupts the CPU's current mode of execution many times a second. The frequency of these interruptions is hardware-dependent; 100 times per second is a typical rate. The kernel handles each clock interrupt by invoking the *clock interrupt handler*. This handler in effect drives the system by performing operations crucial to the system's scheduler. It is in this handler that the scheduling subsystem determines whether or not it is time to schedule a new process for execution.

As it manages resident memory so that processes share the resource equitably, the kernel manages processes' access to the CPU, using a mechanism called *time slicing*. When the kernel switches context from one process to another, it assigns the new process a time-slice, or *quantum* of time, in which to execute. As the process executes, the kernel decrements the quantum. The kernel decrements the quantum with each click from the hardware's clock. The process may relinquish the CPU at any point within its quantum. The process may finish execution and exit, or, if it needs access to resources that are not immediately available (for example, data from a file), the process may put itself to sleep to wait for the resource. If the process continues to execute until its quantum expires, the kernel attempts to schedule another process for execution.

2.6.5 Kernel Daemons

There are a number of system operations that the kernel must actively perform, such as the replacement of pages in resident memory. The kernel's pageout, swapout, and swapin daemons perform page replacement operations as needed. Typically these daemons, which are independent threads of control that execute exclusively in kernel mode, sleep until they are needed. For example, if a process needs to fault in a page of data and there are few pages available for allocation, the page fault will awaken the pageout daemon, which will then pageout data to free pages that can then be reallocated.

Chapter 3

Overview of the Mach Technology in OSF/1

This chapter discusses the core services component of the OSF/1 kernel. This component is derived from Carnegie-Mellon University's Mach technology. The core services component includes the kernel's scheduler and memory management subsystems, both of which provide functionality not generally associated with earlier UNIX systems.

The Mach technology implements a small set of primitive operating system objects, including *tasks*, *threads*, and *memory objects*. These objects can be used to support many different operating system types. In OSF/1, they are used to implement UNIX.

This chapter provides an overview of the Mach technology and the objects that technology provides. Chapters 4, 5, 6, and 7 describe how that technology is used within the OSF/1 operating system.

3.1 Tasks and Threads

A traditional UNIX process is a single entity that encapsulates a set of system resources (memory resources, open files, and so on) and a single thread of control that executes in the context of the set of system resources. In Mach, the process abstraction is split into two separate abstractions: the *task* and the *thread*. A task is a set of system resources that includes a protected virtual address space. A task is not an executable entity; it is merely an environment in which one or more threads can execute. However, this book frequently refers to tasks as though they are executable entities. This is merely a convention.

A thread is a unit of computation that executes within the context of a task. It has access to all of the system resources assigned to the task. If the task contains multiple threads, all such threads share the task's resources.

Each thread has an *execution state* and a *computation state*. A thread's execution state specifies whether or not the thread is executing or can be scheduled for execution.

A thread's computational state specifies its hardware context, including its program counter, its stack pointers, and the contents of hardware registers. When the thread is executing, its computation state is maintained in the CPU's hardware registers. When the thread is not executing, this state resides in the thread's process control block.

A thread's execution and computational state are private. In a multithreaded task, a thread's execution state may differ from the execution state of the other threads, and its computational state usually differs from that of the other threads. A traditional UNIX kernel schedules processes; in Mach and in OSF/1, the kernel schedules threads. Chapter 5 describes the scheduling subsystem in detail.

Tasks and threads are low-level objects. Although the Mach kernel (and OSF/1) provide a set of system calls that can be used to create and manipulate tasks and threads, user-level application developers generally do not work directly with these objects.

Tasks and threads can be used to support different programming models. These models include:

- Non-UNIX parallel programs
- Traditional UNIX processes, implemented as single-threaded tasks
- UNIX processes that contain multiple threads

In OSF/1, tasks and threads are used to implement both single-threaded and multi-threaded UNIX processes.

3.1.1 The task Data Structure

The kernel maintains a **task** data structure for each currently active task. This data structure includes the following elements:

map A pointer to the task's virtual address map. The kernel uses this pointer to access the address map during page fault handling. Address maps are discussed further in Section 3.4.1.

thread_list A list that contains the threads associated with the task.

ipc_translations
Specifies the task's port name space.

3.1.2 The thread Data Structure

The kernel maintains a **thread** data structure for each currently active thread. This data structure includes the following elements:

pcb A pointer to the thread's process control block.

state Specifies the thread's execution state.

The data structure also maintains information used by the scheduling subsystem to schedule the thread for execution.

3.2 The Mach Interprocess Communication Subsystem

Mach implements a message-passing facility that allows tasks to communicate with one another. This facility is referred to as the *Mach Interprocess Communication Subsystem* (*Mach IPC*). (Actually, the facility implements communication between tasks, not processes, but the term "IPC" persists for historical reasons.)

Typically, the IPC subsystem is used to pass data between separate tasks, but it may also be used to pass data between threads within the same task. All IPC operations are managed and secured by the kernel; a task cannot send data to or receive data from another task unless the task has acquired the right to do so.

The IPC facility is based on two abstractions: the *port* and the *message*. The message contains the data being passed, and the port is the means of transferring the message.

3.2.1 Ports

A port is a communications channel that is protected by the kernel. A port is implemented as a message queue within the kernel's address space. When a task sends a message to a port, the kernel copies the message into the kernel's address space and places the message at the end of the port's message queue. When a task attempts to receive a message, the kernel removes the message from the head of the port's queue and copies it out to the task's address space.

Each port has a receiving task, which receives messages sent to the port. Only one task can receive messages from a given port at any time. A port's receiving task can be changed, but the kernel never allows any port to have multiple receiving tasks. A port can receive messages from one or more tasks.

A task can create a new port by issuing a call to the **port_allocate( )** system call. The kernel creates the port by allocating a new message queue within its address space.

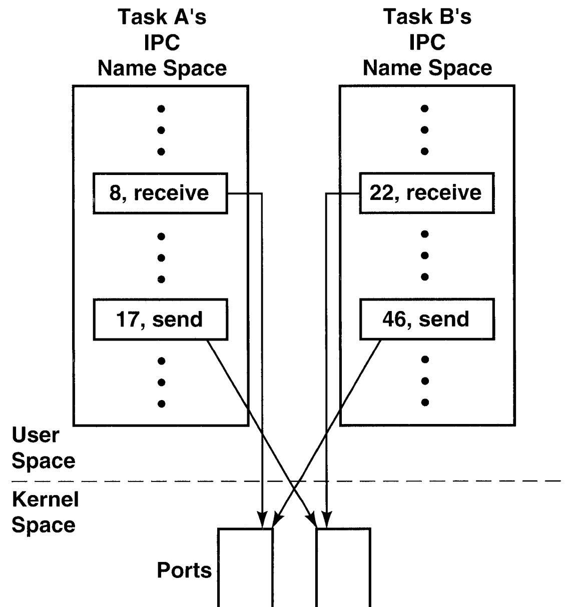
3.2.1.1 Port Rights

A task can use a port only if it has *port rights* to the port. A task can have *send rights*, *receive rights*, or both rights simultaneously.

Each task has a private area in which the kernel stores the names of the ports the task has rights to. This private area is the task's *port name space*. Port names are integers. When a task acquires a new port right, the kernel inserts a name for the right in the task's port name space, giving the right a name that is currently unused.

Port names are local to individual tasks. Other tasks may reference the same port using different names. Figure 3-1 shows how port names correspond to the actual port data structures that reside in the kernel's address space. Tasks **A** and **B** communicate with each other through two ports. **A** receives messages from **B** on the port corresponding to **A**'s local name **8**; **B** sends messages to this port using its local name **46**. **A** sends messages to **B** on the port corresponding to A's local name **17**; **B** receives the messages from this port using the local name **22**.

Figure 3–1. Tasks Sharing Access to Ports Using Private Port Rights



3.2.1.2 Acquiring Port Rights

A task acquires port rights as follows:

- It inherits them from its parent task.
- It receives them in messages sent from other tasks.
- It receives them from the kernel upon issuing calls to **port_allocate( )**.

Ports are used to represent system objects such as tasks, threads, and memory objects. When a task creates of one of these objects (by issuing a system call), the kernel gives the task access to the object by providing the task with rights to the port representing the object. For example, if a task issues a call to **thread_create( )** to create a new thread, the kernel creates the new thread, creates a port to represent the new thread, and gives the task access to the new thread by providing the task with rights to the thread's port.

3.2.2 Messages

Data is passed through ports in messages. The amount of data passed in a message is arbitrary; a message may transfer a byte of data, or it may pass a task's entire address space. Messages can also be used to transfer port rights between tasks.

3.2.2.1 Out-of-Line Data and Lazy Evaluation

Historically, transferring large amounts of data between processes has been an expensive operation because it involves physically copying the data from one address space to another. In Mach, however, such operations are inexpensive because the data is *virtually* copied, not physically copied. When out-of-line data is passed between tasks, the kernel allows both tasks to map the data into their address spaces. The data is mapped *copy-on-write*; if either task attempts to write the data, the kernel copies the portion of data being written to a new page frame before allowing the write operation to proceed.

Allowing tasks to share mappings to out-of-line data is one aspect of Mach's philosophy of *lazy evaluation*. This philosophy can be summed up as follows: defer performing an operation until it absolutely must be performed in the hopes that it may never need to be performed. With respect to passing data out-of-line, the kernel defers physically copying the data until one of the tasks writes the data. If neither task writes the data, it never needs to be copied.

The IPC subsystem passes large amounts of data efficiently because it is tightly integrated with the virtual memory subsystem. Chapter 6 describes in more detail how the virtual memory subsystem implements the copy-on-write mechanism.

3.2.2.2 The Message-Passing Primitives

The common IPC operations are as follows:

- A task sends a message to another task without expecting a reply.
- A task sends a message to another task expecting a reply, but does not wait for the reply. The task receives the reply asynchronously.
- A task sends a message to a task and waits for a reply before continuing with its execution. It receives the reply synchronously.
- A task receives a message from another task.

The Mach kernel exports a set of interfaces that allows tasks to initiate these operations. A task inititiates either of the first two operations by calling the **msg_send**( ) primitive. If the task expects a reply, it uses **msg_receive**( ) to obtain the reply. A task can use this call at any time to receive messages that are queued on the port. A task inititiates synchronous communication by calling **msg_rpc**( ).

3.2.3 Ports as Objects

A server task can use ports to represent the objects it manages, and a client task can invoke an operation on an object by sending a message to the port representing the object. Interactions of this type are referred to as *remote procedure calls* (RPCs).

The Mach kernel is itself a server that creates and manages objects such as tasks and threads. Each of these objects is represented by a port; the system calls that manipulate them are actually remote procedure calls.

Because they are created by the kernel and reside within the kernel's address space, ports are a secure mechanism for providing object references. When a server task creates an object that is to be represented by a port, it must ask the kernel to create the port. When creating the port, the kernel gives receive rights only to the server. The server then controls access to the port. If a client task wants access to the object, it must send a request to the server. If the server chooses to honor the request, it forwards send rights by passing a message back to the client.

3.3 Memory Objects

A *memory object* is an entity that represents a range of virtual pages. A task's address space is implemented by an address map that maps ranges of the address space to specific memory objects. For example, a task that constitutes a UNIX process maps three memory objects into its address space: one that represents the pages containing the text and initialized data from the executable file, one that represents the zero-fill pages containing uninitialized data and heap, and one that represents the zero-fill pages containing the user stack.

Until recently, UNIX systems did not allow processes to map arbitrary objects into their address spaces, and those UNIX systems that do allow such mapping operations restrict them to kernel-defined objects such as files. The memory object abstraction allows developers to implement paging managers that manage application-specific memory objects. These objects may represent files, shared libraries, and databases.

Memory objects are managed by separate tasks called *memory managers* or *paging managers*. A paging manager is responsible for performing pagein and pageout operations on the memory objects it manages.

Each memory object is represented by a port, and the kernel invokes paging operations on the memory object by sending a message to the paging manager on the memory object's port. For example, suppose that a task attempts to execute an instruction whose page is not in resident memory. The kernel's page fault handler requests pagein of the text by sending a message to the paging manager responsible for the task's text memory object.

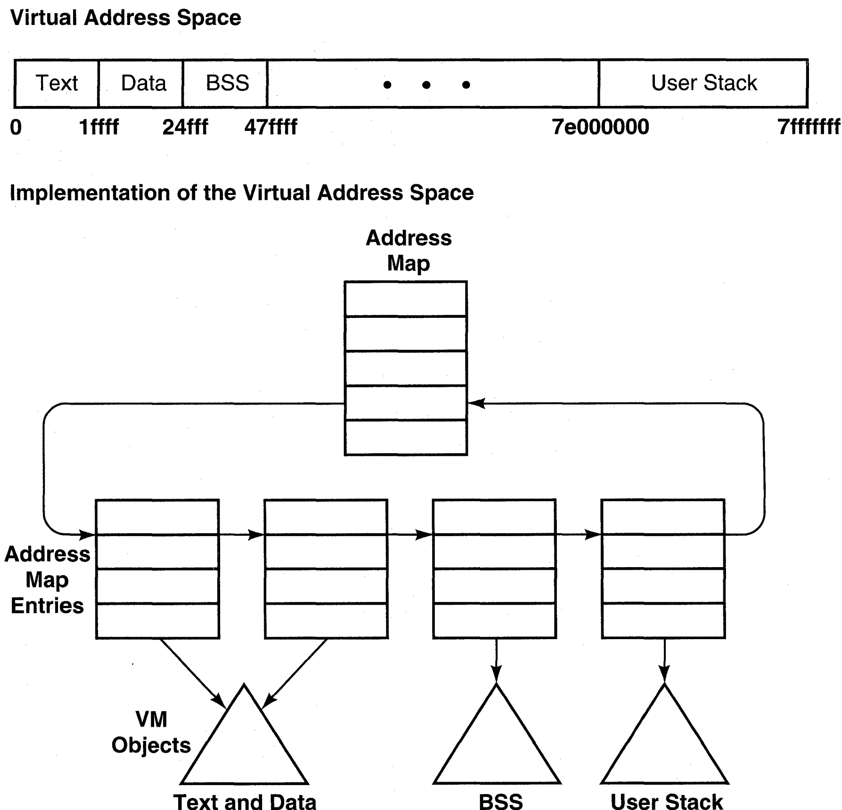
3.4 Mach Virtual Memory Management

The Mach virtual memory management system provides advanced functionality such as copy-on-write operations, mapping of files and application-defined objects, and support for large, sparsely filled virtual address spaces. It is also easy to port.

In Mach, each task is provided with a protected virtual address space that is limited in size only by the addressing capabilities of the underlying memory management unit (MMU). For example, a MIPS R3000 processor supports user address spaces two gigabytes in length. In the MIPS R3000 architecture, a task's address space is that length.

A task's address space is implemented via a hierarchy of machine-independent data structures. A task's address space contains *regions* of allocated memory. An *address map* maps these regions to *virtual memory objects* (*VM objects*). (Section 3.4.1 describes address maps in detail.) Figure 3-2 depicts a sample virtual address space and the address map and VM objects that implement the address space.

Figure 3–2. Implementation of a Mach Virtual Address Space



3.4.1 Task Address Maps

A task's address map is made up of a linked list of *address map entries*, each of which maps a range of virtual addresses to a VM object (or in some instances, to a portion of a VM object). The address map maintains address map entries only for allocated regions of virtual memory; unallocated regions do not have address map entries. In this way, the address map supports the compact representation of sparsely filled address spaces.

A task can map memory objects at arbitrary locations within its address space. A task may use its address space compactly; that is, mapping its memory objects to contiguous ranges of virtual memory. Traditional UNIX processes use compact virtual address spaces. Conversely, a task may use its address space sparsely by mapping its memory objects at widely separated locations in the address space.

Large, sparsely filled address spaces are useful in the implementation of multithreaded tasks. Each thread requires its own user stack, and the task can prevent user stacks from overlapping one another by placing them at widely separate locations throughout the virtual address space.

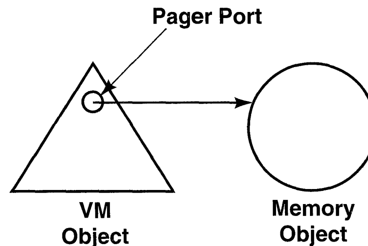
In OSF/1, the program loader allows processes to map shared libraries into their address spaces. This is another application of large, sparsely filled address spaces. See Chapter 8 for a discussion of the loader.

3.4.2 Virtual Memory Objects and Memory Objects

The kernel keeps track of the physical location of a region's pages with the region's VM object; a region's VM object represents the set of pages that are mapped to the region. A VM object allows the kernel to determine whether or not a given page is resident in memory. When a task generates a page fault trying to reference a nonresident page, the kernel uses the underlying VM object to locate the missing page in secondary storage.

A memory object represents a region's pages as they exist in secondary storage. For example, if a region's pages contain executable text, the associated memory object is the file on disk that contains the text. Memory objects are so named because the kernel pages in data from them in response to page faults. A VM object references its memory object through an IPC port called the pager port. See Figure 3-3.

Figure 3–3. A VM Object and Its Memory Object



It is important to distinguish between VM objects and memory objects. The VM object represents pages that exist in main memory and in secondary storage. The memory object represents pages that exist only in secondary storage. For example, when a task generates a page fault within a region of its address space, the page fault handler identifies the VM object that is mapped to the region and initiates a pagein operation by sending a message on the VM object's pager port.

Multiple tasks can map the same VM object into their respective virtual address spaces. For example, consider two separate but concurrent tasks that execute the same program. They both execute from the same text, and the set of pages containing this text is managed by a single VM object. Consequently, both tasks map this object into their address spaces.

3.4.3 VM Object Types

VM objects are of two types. *External VM objects* represent permanent data, and *internal VM objects* represent temporary data. Permanent data exists in secondary storage on a persistent basis. Temporary data is data that is created by a task in memory and does not persist after the task is terminated.

For example, consider the VM objects that are mapped to a task that implements a standard UNIX process. The text VM object represents a set of pages that are stored in a program's executable file. The data in this file is permanent; it will remain in secondary storage after the process's thread completes execution and disappears. VM objects that represent permanent data are referred to as external VM objects because they contain data that

has its origin externally, in permanent secondary storage. Unlike the text VM object, the stack VM object represents data that is temporary and does not have a permanent file in secondary storage. When the process disappears, the data in its stack will disappear as well. VM objects that represent temporary data are referred to as internal VM objects because they contain data that is generated internally by the thread or threads executing in the process.

3.4.4 Memory Objects and Memory Managers

There are two memory object types: *user-initiated* and *system-initiated.*

User-initiated memory objects are created at the request of, or for the benefit of, user-level tasks. The text memory object is a typical example of this type. A task's executable text is stored in an executable file somewhere in the file system. The file system has an associated memory manager to handle memory requests to and from the files it maintains. When the system sets up the task's virtual address space, it asks this memory manager to find the required executable file and create an associated memory object. The memory manager does so and returns a port in a reply message that can be used to access the new memory object (the memory object port). The kernel creates a VM object to represent the text data and inserts the memory object port into the VM object's data structure. The system can then use the VM object to access the pages of the object file through the saved port.

System-initiated memory objects are created in response to requests by the VM system. They correspond to internal VM objects and are created and managed by the system's *default pager*. The kernel creates these memory objects only when the VM system needs to move temporary data out of main memory to free up resources. In OSF/1, the kernel's default pager is referred to as the *vnode pager*. (See Chapter 7, Section 7.2 for more information on the vnode pager).

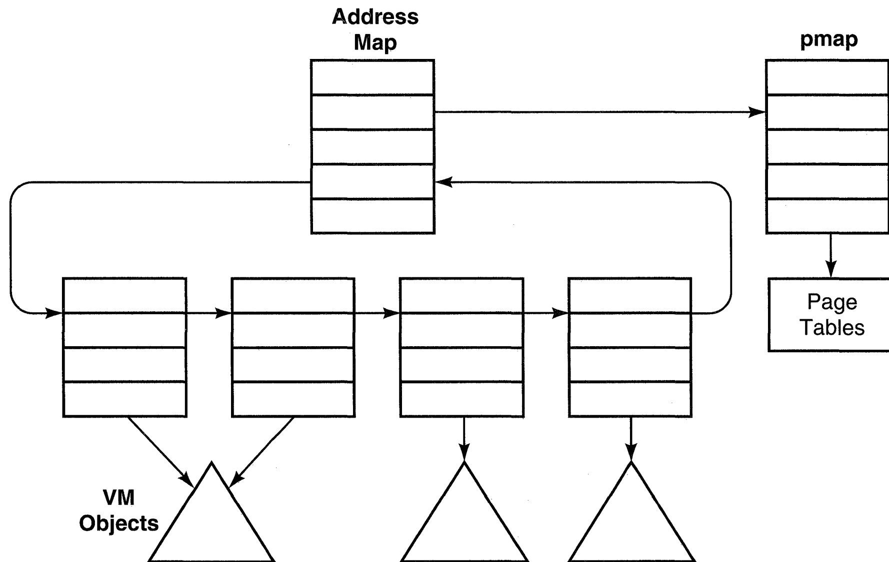
The default pager also serves as the kernel's backup paging mechanism. Because external memory managers execute as separate tasks in user space, the kernel cannot depend upon a given memory manager to page out data on a timely basis. If the kernel must free up a page containing data and the memory manager does not respond quickly enough, the kernel sends a pageout request to the default pager. The default pager always frees pages in a timely manner.

3.4.5 Management of Resident Pages

In Mach, the machine's physical memory is managed as an array of fixed-sized page frames. The kernel does not directly manipulate page frames. Instead, the kernel maps logical pages onto the page frames and manages resident memory by manipulating the logical pages. The logical page abstraction allows the kernel to manage resident memory in a machine-independent fashion.

The size of a logical page is configuration-dependent; it either matches the hardware's page frame size or is a power-of-two multiple of that size. In the example depicted in Figure 3-4, the hardware's page frame size is 1 K and the logical page size is 4 K. When a virtual page is cached in resident memory, it requires four page frames.

Figure 3–4. The Mapping of Logical Page to Page Frames



The logical pages are mapped to the page frames when the kernel is initialized. At that time, the kernel determines how many logical pages will

be required, and then allocates a **vm_page** data structure for each logical page. The kernel uses these structures to manage the state of logical pages.

The kernel maintains three paging queues that it uses to manage page replacement operations:

free queue
: Contains **vm_page** structures whose logical pages are currently available for allocation; when the system requires a new logical page for incoming data, it removes the first **vm_page** on this queue.

active queue
: Contains **vm_page** structures whose logical pages contain data that is actively being used by one or more tasks. When the kernel allocates a logical page for a pagein operation, it places the page's **vm_page** at the end of this queue.

inactive queue
: Contains **vm_page** structures that have recently been active but are not currently in use. If a task needs to access the data contained in an inactive page, the system transfers the page's **vm_page** back to the active queue.

There are circumstances in which a logical page is not available for paging operations. Usually, such a page contains data or text that the kernel must have immediate access to. Pages of this type are referred to as *wired* pages. When the kernel *wires* a page into memory, it is removed from the paging queues and is not subject to the page replacement operations.

3.4.5.1 The Resident Page Table

In addition to the paging queues, the kernel maintains a *resident page table* that keeps track of all virtual pages that are currently cached in resident memory. When a virtual page is paged in, the kernel places the corresponding **vm_page** structure in this table.

The table, also referred to as the *virtual-to-physical* table, is implemented as a hash table. The hash function is based on the *object/offset* value of the virtual page, where *object* specifies the virtual page's VM object and *offset* specifies the virtual page's offset within the object.

3.4.6 Physical Maps

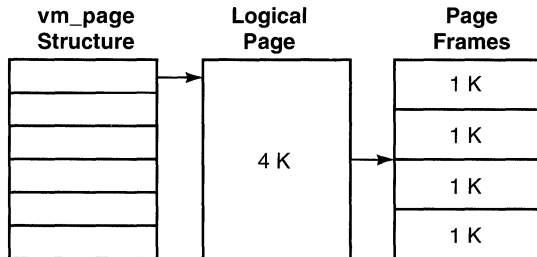
In traditional UNIX, the implementation of virtual address spaces has been, to greater and lesser degrees, tied to the data structures used by the hardware's MMU to perform address translation. For example, 4.3BSD was originally implemented on the VAX; a process's virtual address space was implemented directly by VAX-specific page tables and all of a process's virtual-to-physical translations were maintained in its page tables. The implementation of virtual memory operations such as those performed by the **fork( )**, **exec( )**, and **sbrk( )** interfaces, and the page fault handler was machine-dependent because such operations involved the direct manipulation of page table data structures. Porting 4.3BSD to non-VAX platforms required reimplementing process address spaces and the operations performed on them.

In Mach, the implementation of virtual memory is cleanly separated into machine-independent and machine-dependent parts. The separation is based on the assumption that all MMUs provide a minimum level of functionality. This functionality includes support for separate virtual address spaces composed of fixed-length pages, with each address space described by one or more memory-mapping data structures (page tables). Each page in an address space can be mapped and protected separately.

In Mach, the hardware-dependent memory mapping data structures are represented by the task's *physical map* (pmap). The pmap data structure points to the MMU-specific data structures.

Because all virtual-to-physical translations are maintained in the machine-independent data structures, a task's pmap serves as a cache of those translations. The kernel caches a given translation in the pmap only when the translation is required. The kernel can discard pmap translations at any time because they can always be restored from the task's address map and memory objects. This gives the kernel great flexibility in managing that portion of its address space that it uses to maintain pmaps. Figure 3-5 shows the relationship between the task's address map and VM object data structures, and the pmap data structure.

Figure 3–5. Relationship Between an Address Map and Its Pmap



The kernel manipulates pmaps only when it is absolutely necessary. For example, when a task maps a new memory object into its address space, the kernel does not update the task's pmap until the task actually attempts to reference the memory object's data. The deferring of pmap operations until such operations are necessary is in keeping with Mach's philosophy of lazy evaluation.

Most VM operations are implemented with machine-independent code. When the machine-independent VM performs an operation that must be reflected in a task's pmap, it issues a call through the *pmap interface* to the kernel's *pmap module*. The pmap interface specifies the set of pmap management operations that are required by the machine-independent VM. The pmap module implements these interfaces. The pmap module contains all of the VM's machine-dependent code. Porting the VM system requires little more than reimplementing the pmap module for the new hardware platform.

3.4.7 Mach Virtual Memory Interfaces

The native Mach kernel interface includes a set of primitives that allow tasks to manipulate their virtual address spaces. These interfaces may or may not be available on a given OSF/1 system; their availablity is configuration-dependent.

3.4.7.1 Inheritance of Regions

Like UNIX processes, tasks inherit their virtual address spaces from their parent tasks. However, UNIX processes cannot control how their address spaces are inherited; a process always inherits all of its parent's address space. Mach allows tasks to control the inheritance of address space. A task can pass all or some or none of its address space to its children tasks.

Child tasks can inherit copies of regions, or can actually share regions with their parents. If a child task inherits a copied region, the parent task will not see any of the child's modifications to the region. If the region is shared between the parent and child, both tasks will see modifications made by the other.

3.4.7.2 Protection of Regions

A task can allow read, write, and execute access to any of its allocated regions. The protection attribute has two values: the current value specifies the region's current protection level, and the maximum value specifies the region's maximum protection level. The current protection value can never exceed the level specified by the maximum protection value.

3.4.7.3 Allocation of Virtual Memory

Mach provides two interfaces that a task can use to allocate new regions in its address space. A task can use the **vm_allocate( )** primitive to allocate zero-filled regions of virtual memory, and the **vm_map( )** primitive to map a region of supplied data (memory object) into the task's address space.

The **vm_allocate( )** routine maps internal VM objects into the virtual address space, while the **vm_map( )** routine maps external VM objects into the address space. Before a task can map a memory object using **vm_map( )**, it must have acquired access to the object from the object's memory manager. Tasks may remove regions from their address spaces by using the **vm_deallocate( )** primitive.

3.4.7.4 Region Management Interfaces

When a task allocates a region of its address space, the VM system sets default protection and inheritance values. The task can change these values by calling the **vm_protect**( ) and **vm_inheritance**( ) primitives, respectively.

The **vm_region**( ) primitive provides a means by which a task can request information about allocated regions in another task's address space. This information includes the following:

- The region's current and maximum protection values
- The region's inheritance value

A task must have access to the port representing the other task to use the **vm_region**( ) primitive.

The VM interface also provides primitives that allow a task to read, write, and copy from another task's address space. Assuming that the target task is accessible and the region's protection value is set appropriately, a task can use the **vm_read**( ) primitive to read from the target region, the **vm_write**( ) primitive to write to the target region, and the **vm_copy**( ) primitive to copy the target region.

3.4.8 Memory Managers and the External Memory Management Interface

In traditional UNIX systems, the virtual memory management system allows tasks to map only system-defined objects, such as files, into their address spaces. The code that implements the creation and management of such objects is embedded in the kernel. Consequently, extension of memory mapping functionality requires developers to modify, rebuild, and retest the kernel.

In OSF/1, the memory mechanism is implemented through memory managers that are not embedded in the kernel; any user-level application can use the mechanism of the memory object to provide data to client programs by allowing those clients to map the data directly into their address spaces. The flexibility of OSF/1's VM system supports the

development of complex virtual memory applications such as transaction and data management systems. These applications can be developed and tested without modifying and rebuilding the kernel.

Mach defines an *external memory management interface* (EMMI) that allows the kernel and memory managers to interact with one another in the management of virtual memory.

The EMMI is actually two sets of routines: those implemented by memory managers to be used by the kernel, and those implemented by the kernel to be used by the memory managers.

Each memory manager implements a *memory object interface*, a set of routines that the kernel uses to issue requests to memory managers. For example, the kernel pages in a memory object's data by issuing a call to the memory manager's **memory_object_data_request**( ) routine; the kernel pages out out a memory object's data using the memory manager's **memory_object_data_write**( ) routine.

The kernel implements the *cache management* interface, which is the set of routines that memory managers use to handle pagein and pageout requests from the kernel and to control access to pages that are cached in resident memory. For example, a memory manager that allows separate hosts to share read/write access to its objects can serialize modifications to an object by making the pages cached in one kernel read-only while a task on another machine writes the pages.

For a more detailed discussion of the external memory management interface, see Chapter 7, Section 7.7.2.

Chapter 4

Processes: Structure and Management

OSF/1 provides an execution environment for UNIX processes that implements all of the features provided by traditional UNIX systems. These features include the well-known process management system calls (such as **fork**( ), **exec**( ), **exit**( ) and **wait**( )), an implementation of the signal facility, and an implementation of the job control facilities.

Providing support for multithreaded processes and making the traditional UNIX facilities function properly in multiprocessor environments has had a profound impact on the design and implementation of OSF/1's execution environment.

4.1 Process States and Data Structures

Traditional UNIX maintains a process's state using a set of data structures that includes the following:

proc Structure

Encapsulates state that must remain in resident memory at all times. For example, a process's scheduling state is maintained

in the **proc** structure because the scheduler needs to update this state whether or not the process itself is resident.

user Structure
Can be swapped out to secondary storage and encapsulates state that needs to be resident only when the process is executing. For example, a process's *file descriptor table*, which the process uses to access the files it has opened, is maintained in the **user** structure because it is not needed when the process is swapped out.

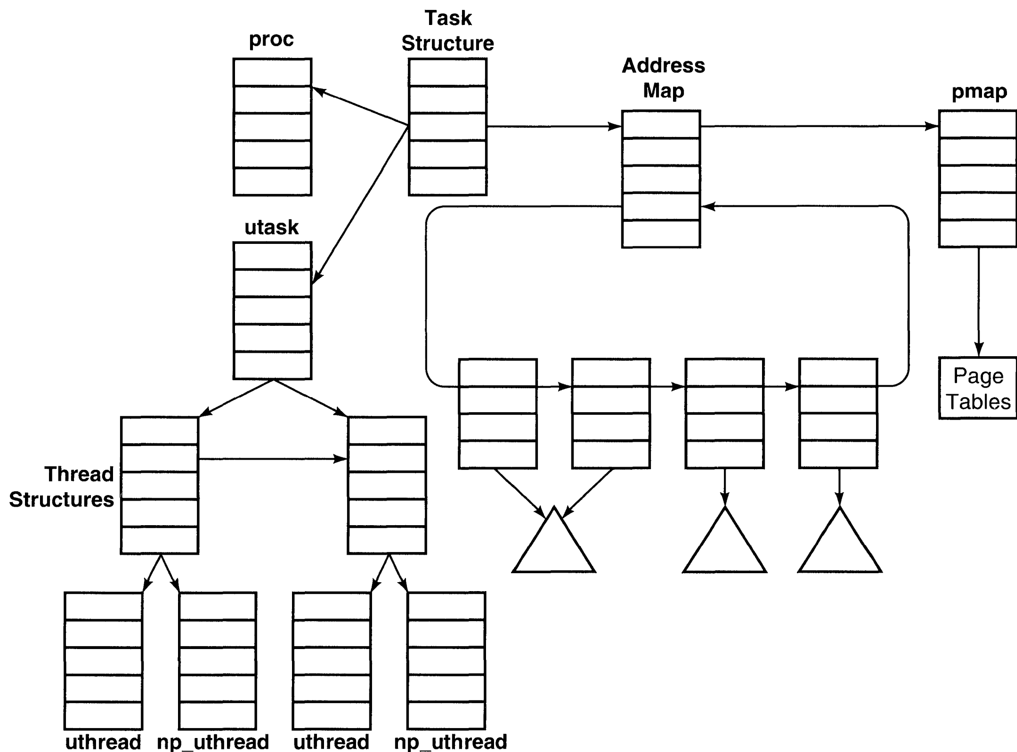
Memory Map Structure
Implements the process's virtual address space.

The **proc** structure contains a pointer to the process's memory map and the **user** structure contains a pointer to the **proc** structure.

The structure of an OSF/1 process differs significantly from its traditional counterpart. In OSF/1, a process consists of a task and one or more threads; consequently, much of a process's state is maintained in its *task* and *thread* data structures. The **user** structure has been changed significantly in OSF/1; it has been split into three separate structures: the **utask** structure, the **uthread** structure, and the **np_uthread** structure.

Figure 4-1 illustrates a process as implemented in OSF/1. In this example, the process has two threads and so has two thread data structures. As shown in the figure, the task structure contains pointers to the process's address map and the thread data structure. The task structure also contains a pointer to the **utask** data structure, while the thread structure contains pointers to the **utask**, **uthread**, and **np_uthread** structures. The **proc** structure contains pointers to the task and **utask** structures.

Figure 4–1. Structure of a Process in OSF/1



4.1.1 The proc Structure

In OSF/1, much of the state contained in the traditional UNIX **proc** structure is maintained in the process's task and thread structures. For example, a traditional UNIX kernel accesses a process's memory map through the **proc** structure, while the OSF/1 kernel accesses a process's memory map through the task structure. In traditional UNIX, a process's scheduling state is maintained in the **proc** structure, while this is not the case in OSF/1. Because a process may have more than one thread of control, it may have more than one schedulable entity. The OSF/1 kernel schedules threads, not processes; and so a process does not have scheduling state, its threads do. Each thread's scheduling state is maintained in the thread's data structure.

The OSF/1 **proc** structure contains the following state information:

- The process's identifiers, including its process ID number and the parent process's ID number, and its effective and real user and group ID numbers.
- The pointers to the **proc** structures associated with the process's parent and sibling processes.
- Some of the process's signal state. Signals are discussed in Section 4.4.

4.1.2 The user Structure

In OSF/1, a process's executable entity is its thread, and its system resource entity is its task. To support the division of the process abstraction into the task abstraction and the thread abstraction, OSF/1 divides the **user** structure into three separate structures: the **utask** structure, the **uthread** structure, and the **np_uthread** structure. The **utask** structure maintains state that is task-specific, while the **uthread** and **np_uthread** structures maintain state that is thread-specific.

If a process contains multiple threads instead of the standard single thread, the traditional **user** structure contains state that should be shared among the threads (for example, the file descriptor table) as well as state that should be private to each thread (for example, the state of a thread's kernel stack). In OSF/1, however, the **utask** structure is used to manage the state of shared resources, and the **uthread** structure is used to manage the state of private resources. Each process has a single **utask** structure and as many **uthread** structures as there are threads in the process. If the process is a standard one, it contains one thread and, therefore, one **uthread** structure.

4.1.2.1 The utask Structure

The **utask** structure maintains those fields of the traditional user structure that are related to system resources. This includes the file descriptor table, the process's file creation mask, and any environment variables the process might have inherited or set. The **utask** structure also specifies the sizes and starting addresses of the process's regions of virtual memory, and includes fields used to manage process-specific signals. OSF/1's implementation of signals is discussed in Section 4.4.4.

4.1.2.2 The uthread and np_uthread Structures

The **uthread** and **np_uthread** structures maintain states that are thread-specific. The **uthread** structure includes the directory search file name cache. The **np_uthread** structure includes fields used to manage signals related to program exceptions (see Section 4.4.6 for a full discussion). The difference between the structures is that the state maintained in the **uthread** structure may be paged out, while the state maintained in the **np_uthread** structure must remain resident.

4.2 Allocation of proc Structures

In contrast to traditional UNIX systems, which statically allocate the system's supply of **proc** structures during system initialization, OSF/1 minimizes the amount of memory that is statically allocated for process management by allocating **proc** structures on demand. The kernel statically allocates a table of pointers to **proc** structures, but the structures themselves are allocated dynamically. This scheme allows processes to be found by index operations into a static table, while also providing memory requirements that scale well as the number of active processes rises.

4.3 The Process Management System Calls and Multithreaded Behavior

Multithreaded processes have notable implications for the standard process management system calls **fork( )**, **exec( )**, **exit( )**, and **wait( )**.

For example, when a thread within a multithreaded process executes a call to **fork( )**, it could duplicate all threads in the parent process or just the thread that issued the system call. In OSF/1, **fork( )** duplicates only the calling thread.

Another issue that must be addressed with respect to multithreaded processes has to do with the synchronization of process management calls. For example, it is possible that one thread may call **fork( )** at the same time that another thread is terminating the process with a call to **exit( )** or overwriting the process's address space with a call to **exec( )**. To preserve the integrity of the process and ensure consistent behavior with respect to the process management calls, OSF/1 provides a mechanism that allows the system calls to synchronize with one another. This mechanism is implemented through fields in the **proc** data structure that are used to record calls to **exit( )** and calls to **fork( )**.

When a thread calls **fork( )**, the **fork( )** system call checks to see if another thread has already called **exit( )**. If **exit( )** has been called, **fork( )** suspends the current thread to wait for the process to be terminated.

If another thread has not called **exit( )**, **fork( )** can proceed with its operation. Before it does so, it increments a field in the **proc** structure to record that a **fork( )** operation is underway. The **exit( )** and **exec( )** system calls check this field to synchronize their operations with **fork( )** operations; neither **exit( )** or **exec( )** are allowed to proceed until all current **fork( )** operations are completed.

The **wait( )** system call is used by a parent process to wait for a child process to exit. OSF/1's implementation of this system call allows only one thread in a parent process to wait for a child to exit. If a thread issues a call to **wait( )** on a child that is already being waited on, the call will fail.

4.4 The Signal Facility

The signal facility is one of the most complicated features of the UNIX system. Originally a mechanism for terminating misbehaving processes, the signal facility, while retaining its original functionality, has evolved into a somewhat primitive medium for interprocess communication and process execution management.

For example, the job control facility uses the signal facility to allow users to switch which process or group of processes has access to the terminal. A user can suspend the execution of the process or process group that is executing in the foreground by entering the Suspend character (commonly <**Ctrl-Z**>) from the keyboard, and resume the suspended "job" at some later time by entering the **fg** command at the shell. Entering a Suspend character sends a **SUSPEND** signal to the job; entering the **fg** command causes the shell to send a **CONTINUE** signal to the job.

A process may receive a signal by generating a program exception (dividing by zero, or referencing an invalid address), or it may receive a signal from an external source (from another process, for example, or from the terminal when a user enters the Kill character or the Suspend character). Signals caused by exceptions are called *synchronous* signals; signals originating externally are called *asynchronous* signals.

Included among the traditional set of signals are the following:

SIGSEGV The process generated a segmentation violation exception.

SIGILL The process attempted to execute an illegal instruction.

SIGBUS The process generated a bus error exception.

SIGHUP The terminal line associated with the process has been hung up.

SIGALRM A timer that was set by the process has expired.

SIGSEGV, **SIGILL**, and **SIGBUS** are synchronous signals; **SIGHUP** and **SIGALRM** are asynchronous signals.

Processes can post signals to one another with the **kill**() system call. For example, the following line of code sends a hangup signal to the process specified by *pid*:

kill(*pid*, SIGHUP);

Unless the target process (the process specified by *pid*) has installed a *signal handler* (see Section 4.4.2.2), the process will terminate when it receives the SIGHUP signal.

4.4.1 The Posting of Signals

Regardless of a signal's source, the target process must execute in order to receive the signal. The target process cannot explicitly check for pending signals; the kernel checks for pending signals each time the process transitions from kernel mode to user mode.

This is what happens when a traditional UNIX process generates a program exception (note that this is not what happens when an OSF/1 process generates a program exception; see Section 4.4.5 for more information):

1. The process generates the exception, thereby invoking the kernel's trap handler. The CPU is now executing in kernel mode, but within the context of the process.
2. The trap handler diagnoses the problem and posts a signal to the process.
3. Before the trap handler returns the CPU to user mode, it checks to see if the current process (the one that generated the exception) has a signal pending delivery, and of course there is such a signal because the trap handler just posted it.
4. The trap handler calls the kernel's signal delivery routine and the signal is delivered.

The posting of signals from external sources is somewhat more complicated, but the target process still must transition from kernel mode to user mode in order to receive the signal.

4.4.2 Signal Delivery

What happens when a signal is delivered to a process depends on the signal's type and on the process's *disposition* with respect to that signal type. A process's disposition to a signal type specifies how the process will respond upon receiving that signal. A process may respond to a given signal in any of the following ways:

- Perform the default action associated with the signal
- Ignore the signal
- Catch the signal with a signal handler

4.4.2.1 Default Actions

UNIX systems specify a default action for each supported signal type. A signal's default action may be one of the following:

- Discard the signal and do nothing. For example, when a child process terminates execution, the kernel notifies the parent process by sending it a **SIGCHLD** signal, which the parent, by default, ignores.
- Terminate the process. For example, the kernel sends the **SIGKILL** signal to all active processes when the system is about to be shut down. This signal forces each process to exit.
- Terminate the process and produce a *core file* that contains the in-core image of the process at the time it received the signal. Core files are useful for debugging purposes. For example, a process that generates a segmentation violation will receive a **SIGSEGV** signal, which by default terminates the process and produces a core file. A programmer may then examine the core file to locate the programming error responsible for the segmentation violation.
- Suspend the process's execution. For example, when a user enters the Suspend character at the terminal, the terminal driver sends a **SIGSTOP** signal to the foreground process, by default causing the process to suspend execution.

4.4.2.2 Nondefault Actions

A process may choose not to perform a signal's default action. Instead, a process may choose to ignore the signal or catch the signal with a signal handler.

A signal handler is a routine specified by the application that allows the application to customize its response to the signal. For example, an application may use a signal handler to clean up state before terminating itself. When the process receives a signal for which it has installed a signal handler, the kernel turns control over to the signal handler code when delivering the signal.

The **sigaction()** system call allows a process to manipulate its disposition with respect to signals. This system call can be used to install signal handlers. The kernel does not allow processes to ignore the **SIGKILL** and **SIGSTOP** signals, or to install signal handlers for these signals.

4.4.2.3 Masking Signals

A process may choose to temporarily mask the delivery of one or more signals. If a process masks a specific signal and that signal is posted to the process, the kernel places the signal in the process's set of pending signals but does not allow the signal to be delivered. If the process subsequently unmasks the signal, the signal will then be delivered. In OSF/1, a process can manipulate its signal mask with the **sigprocmask()** system call.

4.4.3 The Signal System Calls

OSF/1 implements the POSIX compliant set of signal system calls, and provides compatibility libraries for accessing the BSD and System V signal interfaces. The POSIX signal system calls include the following:

sigaction() Manipulates a process's disposition to one or more signals.

sigprocmask()
Changes the process's current signal mask.

sigsuspend( )
Changes the current signal mask and suspends the process until a signal is delivered that either terminates the process or invokes a signal handler.

sigpending Examines any signals that are waiting to be received but are currently blocked.

4.4.4 Implementation of the Signal Facility

OSF/1's implementation of the signal facility differs substantially from the implementation found in traditional UNIX systems. Most of these differences stem from work done to make signals behave properly in multiprocessor environments and with processes that are multithreaded.

4.4.4.1 Signals and Multithreaded Processes

Traditional UNIX processes have a single thread of control so there is no confusion about which of a process's threads should receive a given signal. However, in the case of multithreaded processes, the issue is not so clear.

When the kernel posts a signal in response to an exception, it sends the signal to the thread that generated the exception. The kernel delivers signals that are generated asynchronously to a designated thread within the process. This is the process's *first thread*, the oldest currently active thread within the process.

Because it must distinguish between process-specific and thread-specific signals, the kernel must provide separate places for the posting of process-specific and thread-specific signals. The kernel posts process-specific signals by adding them to the **proc** structure's **p_sig** signal mask. The kernel posts thread-specific signals to the **uu_sig** field of a thread's **np_uthread** data structure. The **uu_sig** field contains a mask of thread-specific signals currently pending delivery.

In OSF/1, asynchronous process-specific signals are still posted through the **psignal( )** routine. If the signal is generated by an exception, it is delivered directly to the thread through the **thread_psignal( )** routine.

4.4.4.2 Multiprocessor Implications of Signal Posting

When OSF/1 is running on a multiprocessor machine, it is conceivable that two simultaneously executing threads may attempt to post a signal to the same process at the same time. The kernel protects a process's signal state from such occurrences by protecting the signal fields within the **proc** structure with a lock; before these fields can be modified, a modifying thread must first acquire the lock.

Traditional UNIX kernels post signals exclusively with the **psignal( )** routine regardless of whether or not the signal is being posted by a process running at base level or by a device driver routine that is executing at interrupt level. If both base level code and interrupt level code are allowed to lock the same data structure, the kernel can be forced into a deadlock, as illustrated by the following example:

1. A process executing at base level posts a signal to another process by using the **kill( )** system call. This system call posts the signal by calling **psignal( )**, which posts the signal, first locking the **proc** structure to synchronize with other signal delivery operations.
2. The CPU receives an interrupt from the tty driver, which wants to post a signal to the same process. The interrupt handler attempts to lock the signal data, but cannot acquire the lock because the process it has interrupted holds the lock.
3. The interrupt handler waits for the lock to be released, and so never returns; the interrupted process cannot release the lock because the interrupt handler does not return. The system is deadlocked.

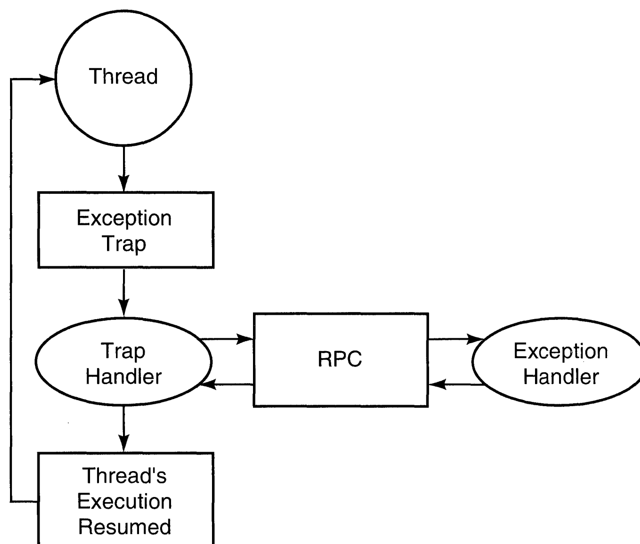
OSF/1 prevents this type of deadlock from occuring by implementing two versions of the **psignal( )** routine. Code that runs at base level calls the **psignal_internal( )** routine, which locks the **proc** structure to post the signal. Code that runs at interrupt level posts signals with the **psignal_indirect( )** routine. This routine places a signal-posting request on a queue that is serviced by a dedicated kernel thread. This kernel thread runs exclusively at base level and can safely lock **proc** structures.

4.4.5 The Exception Handling Facility

In OSF/1, synchronous signals are managed by the kernel's *thread exception* facility. This facility allows processes to customize their responses to exceptions by installing *exception handlers*. The facility is based on an RPC mechanism that is implemented using Mach IPC primitives.

An exception handler is a server task that performs operations in response to remote procedure calls. When an OSF/1 process generates an exception, the trap handler sends an RPC message to the process's exception handler. The handler performs the appropriate operation, and then sends a reply back to the trap handler. The trap handler then begins the transition back to user mode. See Figure 4-2.

Figure 4–2. The Exception Handling Model



4.4.5.1 The UNIX Exception Handler

OSF maintains a default exception handler, the *UNIX exception handler*, which fields all exceptions that are not being caught by application-specific exception handlers. When the UNIX exception handler receives an exception, it converts it to the appropriate UNIX signal and posts the signal with the **thread_psignal( )** routine.

The UNIX exception handler is started during the kernel's initialization and is set to listen on the **init** task's task exception port. Since **init** is the ancestor of all tasks, a given task inherits **init**'s task exception port unless one of its other ancestors has installed a different task exception handler.

4.4.5.2 Design Goals of the Exception Handling Facility

Exceptions can be categorized by type. Some exceptions, such as those associated with invalid memory references, are caused by problems with a program's logic and indicate the need for debugging. Other exceptions may be caused by error conditions that the program can recover from and must be handled by application-specific error handlers. For example, an arithmetic application may, in response to a floating-point underflow exception, use an error handler to substitute 0 (zero) for the underflow value before continuing execution.

Exceptions also play a role in the implementation and use of debugger applications. Interactive debuggers rely on hardware exceptions to implement tracing and breakpoint facilities. When, for example, an executing program reaches a breakpoint, it generates a breakpoint exception that is then intercepted by the debugger. Upon receiving the exception, the debugger allows its user to examine the program's current state.

The design goals of the OSF/1 exception facility are as follows:

1. Provide full support for debuggers and error handlers.
2. Allow error handlers to execute in a context separate from the thread that generates the exception.
3. Support the implementation of sophisticated debugging facilities such as remote debuggers.

Exception handlers can be task-specific or thread-specific. Program debuggers are generally task-specific, while error handlers are generally thread-specific.

Note that exception handlers are distinct from signal handlers. A signal handler executes in response to a signal, while an exception handler performs operations in response to exceptions. The UNIX exception server handles a given exception by sending the corresponding signal to the process. If the process has installed a signal handler for that signal, the signal handler will be invoked when the process receives the signal.

4.4.5.3 Implementation of the Exception Facility

The implementation of the exception facility is based on the client/server paradigm: a thread that produces exceptions becomes a client for one or more exception handler servers. OSF/1 implements the exception handling RPC using three ports: the *thread exception port*, the *task exception port*, and the *thread exception clear port*.

The thread exception port is used for handling thread-specific exceptions; if a thread wants to handle exceptions of this type, it arranges for its exception handler to listen for RPCs on this port. In a multithreaded task, each thread may have its own thread exception port and can install its own exception handler. The same exception may get handled in different ways depending on which thread generated the exception.

The task exception port is used for handling task-specific exceptions. If a task wants to handle exceptions of this type, it arranges for the exception handler to listen for RPCs on this port. This is the port that a debugger would use to handle exceptions.

A thread's exception clear port is the port used by an exception handler to post the return message and complete an exception RPC. Both thread-specific and task-specific handlers post their replies to this port.

Each task inherits its task exception port from its parent task. All threads within the task by default have their thread exception ports set to **PORT_NULL**, but a thread is free to initialize a thread exception port and associate it with a thread-specific exception handler. If a thread has not set up a thread-specific exception handler, any exceptions it generates are sent to the task's exception handler on the task exception port.

The **thread_doexception( )** routine is the trap handler's interface to the exception facility. It executes the RPC to the exception server by sending the server a message on the process's *exception port*. Since all processes share the same exception port by default, the message passes to the UNIX exception server. When the server finishes handling the exception (converting it to a signal and posting the signal to the process), it sends a reply message to **thread_doexception( )** on the victim thread's *exception clear port*. This port must exist before **thread_doexception( )** can execute the RPC; if the thread does not have an exception clear port, **thread_doexception( )** allocates one for the thread.

4.4.6 Signal Handlers

Like traditional implementations of the signal facility, the OSF/1 implementation allows a program to install signal handlers. However, unlike traditional signal facilities, in which all handlers can be considered process-specific, OSF/1 signal handlers can be either process-specific or thread-specific. The distinction here is that process-specific handlers catch asynchronous signals, while thread-specific handlers catch synchronous signals.

In 4.3BSD, the kernel references a process's signal handlers through the *u_signal* field maintained in the process's **user** structure. This field is an array with an entry for each of the signals supported by the system. Each entry specifies a signal's disposition: the action to be taken if the process receives that signal. If the process has installed a signal handler for a given signal, the signal's entry in the *u_signal* array contains a pointer to the signal handler function. When the process receives that signal, the kernel finds the signal handler by indexing into the *u_signal* array.

In OSF/1 the **user** structure is split into the **utask**, **uthread**, and **np_uthread** structures. Both the **utask** structure and the **np_uthread** structure contain a signal disposition array. The **utask** structure's array is in the *uu_signal* field, while the **uthread** structure's array is in the *uu_tsignal* field. The **utask**'s array has entries for those signals that are specific to the process, including those that by default either force the process to suspend or to terminate execution. When the process installs a signal handler for any of these signals, the kernel references the handler through the *uu_signal* field.

The array maintained in the **uthread** structure has entries for all signals that are specific to the thread, namely those signals that are generated by thread exceptions. When a thread installs a signal handler for one of these signals, the kernel references the handler through the *uu_tsignal* field. Note that in a multithreaded process, each thread has its own **uthread** structure, and hence, its own *uu_tsignal* field. Consequently, two threads within such a process could install different signal handlers for the same signal.

The system calls used to install signal handlers automatically place each handler in its proper field. For example, if a thread wants to install a handler to catch a process-specific signal, the kernel places the handler in the **utask**'s *uu_signal* array. If the thread wants to install a handler for a thread-specific signal, the kernel places it in the *uu_tsignal* field contained in the thread's **uthread** structure.

In a multithreaded process, if two threads try to install different signal handlers for the same process-specific signal, the second installation will overwrite the first one. However, as mentioned previously, two threads can install different signal handlers for the same thread-specific signal.

4.4.7 Unix System Calls, the U-area, and Interrupted System Calls

In OSF/1, the UNIX system call mechanism copies system call arguments from the process's user stack directly to its kernel stack. In traditional UNIX systems, these arguments are copied to the process's u-area, not its kernel stack, and the **user** structure includes a field for storing system call arguments. This field does not exist in either the **utask** or **uthread** data structures.

In OSF/1, when a signal interrupts a system call, the call returns control to **syscall**( ) through the regular return mechanism along with an error code indicating the occurrence of an interrupt. If the system call is one that should be restarted, **syscall**( ) adjusts the process's user stack pointer so that the system call will be re-invoked when the process next executes in user mode.

In 4.3BSD, the **user** structure includes a field that specifies which signals can interrupt system calls. This field, the **u_sigintr** field, contains a bitmask that can be set or modified when the process calls the **sigaction**( ) routine to specify action for a given signal. If the signal's bit is turned on in this field,

the process will reexecute system calls that are interrupted by the signal; if the bit is not set, system calls that are interrupted by the signal are aborted and return the value **EINTR**.

OSF/1 retains the **u_sigintr** field within the **utask** structure. Consequently, a signal's disposition as related to interrupted system calls is set on a per-process basis, not a per-thread basis. In multithreaded processes, a thread cannot change a signal's system call disposition without affecting the process's other threads.

Chapter 5

The Scheduling Subsystem

By default, the OSF/1 scheduler allocates the system's CPUs on a timesharing basis; concurrently active threads have approximately equal access to the system's CPU resources. If OSF/1 is running on a symmetric multiprocessor platform, all threads have approximately equal access to all of the CPUs.

Timesharing is based on the programming model of a uniprocessor executing serial programs. Programs that are concurrently active compete with one another for access to the lone CPU, and the scheduler attempts to share the CPU fairly among the programs. In an environment that supports parallel programming models, a timesharing scheduling policy is not always desirable.

Therefore, the OSF/1 scheduler provides mechanisms for supporting the scheduling requirements of various parallel programming models. These include the following:

- Applications that require access to fixed numbers of CPUs.
- Applications that are able to optimize their performance by advising the scheduler which thread or threads to execute next.

This chapter begins by describing how the OSF/1 scheduler subsystem implements timesharing. The chapter then describes how the scheduler supports nontimesharing programming models. The chapter closes with a

discussion of OSF/1's support for CPU-usage timers, which are timestamp-based timers that allow developers to accurately determine how CPUs are being utilized over time.

5.1 Timesharing

The OSF/1 scheduler implements timesharing for both uniprocessor and multiprocessor environments. The design of a timesharing scheduler must address several issues related to the goal of allowing all active processes approximately equal access to the system's CPU resources. To introduce these issues, this section briefly describes the design of the 4.3BSD scheduler, which implements timesharing in a uniprocessor environment.

5.1.1 The BSD Scheduler

In BSD UNIX systems, the scheduler allocates the CPU to each process in fixed-length units of time. This unit is commonly referred to as the *quantum*, and is usually set to 0.1 second. When the kernel switches context to a new process, it resets the CPU's quantum; as a process executes, the clock interrupt handler decrements the quantum.

An executing process may run until its quantum expires (at which time the CPU may be context-switched to another process), or the process may relinquish the CPU before its quantum expires by blocking to wait for an event. If a process has not executed to completion during the quantum, it will be rescheduled for execution.

When it is time to perform a context switch, the scheduler determines which of the currently runable processes to execute next by searching the scheduler's *run queue* for the process with the best *scheduling priority*. A process's priority is related to the amount of time it has used the CPU. When a new process is created, the kernel assigns the process a base priority that is relatively high, so that the process can begin executing as soon as possible. As the process executes, the system increments a counter to record the amount of time the process uses the CPU.

In BSD UNIX, a process's CPU utilization is recorded in the **proc** structure's *p_cpu* field; as a process executes, the clock interrupt handler

increments the *p_cpu* field to record each clock tick. Throughout a process's lifetime, the scheduler adjusts its priority to reflect the amount of time the process has utilized the CPU. Processes that stay on the CPU for the duration of their quanta have lower priorities than processes that block frequently.

Historically, UNIX scheduling policies have favored interactive programs over computation-bound programs. Interactive programs, such as editors, tend to use the CPU for short intervals of time, blocking frequently to perform I/O. Because they spend a lot of time waiting for I/O instead of utilizing the CPU, interactive programs tend to maintain relatively high priorities. On the other hand, computation-bound programs require extended access to the CPU, and therefore often have low priorities relative to interactive programs. If there are several interactive programs active at the same time as a computation-bound program, the computation-bound program will not get extended access to the CPU.

To prevent computation-bound programs from perpetually remaining at low priorities, timesharing schedulers implement mechanisms for elevating the priorities of long-running jobs. In BSD UNIX systems, the scheduler elevates a process's priority using a *usage aging* mechanism. This mechanism causes the scheduler to gradually forget a process's CPU utilization such that the process's priority rises if it has not executed recently.

A process's utilization ages at an exponential rate that is adjusted according to the system's load average. An exponential rate means that usage accumulated in the last minute costs the process n units, usage from the previous minute costs $1/2n$ units, usage from the minute before that costs $1/4n$ units, and so on.

However, an exponential aging rate, by itself, produces an undesirable effect when the system is heavily loaded. Under heavy load, the scheduler has many processes to allocate the CPU to, and each process must wait longer for access to the CPU. Consequently, the scheduler forgets all usage and is no longer able to distinguish those processes that are light consumers of the CPU from those processes that are computation-bound. Under these circumstances, the scheduler needs to slow down the aging rate so that priorities do not improve too rapidly. The scheduler must account for system load as it maintains process priorities.

There are two methods that can be used to factor in the load: use the load to adjust the rate at which a process's utilization ages, or use the load to adjust the rate at which a process accumulates utilization units. The BSD

scheduler uses the age-rate adjustment method. Once a second the BSD scheduler decrements the utilization values of each runable process using the following formula:

$$p_cpu = \left[\frac{2 \cdot load}{2 \cdot load + 1} \right] p_cpu$$

(In the formula, *load* is a sampled average of the number of processes that are waiting in the run queue over the last minute.)

5.1.2 The OSF/1 Scheduler

Unlike the BSD scheduler, the OSF/1 scheduler does not use the load to adjust the aging rate. Instead, the load is used to adjust the rate at which processes accumulate units of utilization.

The OSF/1 scheduler assigns each new thread a base priority, then adjusts the priority throughout the thread's lifetime to account for the thread's utilization of the CPU. To expedite the scheduling of kernel threads that perform operations critical to the performance of the system (the pageout daemon, for example), the scheduler assigns user threads a worse base priority than kernel threads.

5.1.2.1 Accumulating Utilization Units

As a thread executes, the clock interrupt handler increments the thread's utilization counter. By default, this counter, which is maintained in the thread structure's *sched_usage* field, accumulates time using a time-sampling mechanism that charges the current thread with a full clock tick each time the kernel handles a clock interrupt. Charging CPU usage on a time-sampling basis introduces a margin of error because the system cannot accurately determine exactly what happens in the interval between samples. Sampling errors include the following:

- A process that executes for a fraction of the interval between clock interrupts accrues a full tick's worth of utilization if it is executing at the interrupt time.

- Conversely, a process that executes for a fraction of the interval but is not executing when the clock interrupts accrues no utilization.
- The kernel may have been executing interrupt code (on behalf of another process) during the interval, but utilization is still charged to the current process.

Statistical timing discrepancies are generally not large enough to adversely affect the scheduler's algorithm, but other timing applications, such as those that profile a program's execution, can be severely affected by such discrepancies. OSF/1 includes a timestamp-based timing facility that provides a much more accurate means for determining CPU utilization. This facility is discussed in detail in Section 5.4.2. If the underlying hardware provides support for this facility, the scheduler can be configured to use this timer instead of the statistical timer.

Using the statistical timing mechanism, a scheduler charges threads with CPU utilization in units of microseconds. Although the number of microseconds between clock interrupts is fixed, the number of microseconds a thread is charged during that interval depends upon the current load average of the system.

5.1.2.2 The Calculation of the Load Average

The scheduler determines the system's load average using a 2-step calculation: first the current load is calculated, and then that value is exponentially averaged with the previously derived load average. Averaging the current load with the previous load average smooths the impact of abrupt load changes.

The scheduler performs the load average calculation once a second using the kernel's scheduler thread. The scheduler thread calculates the load average according to the following formulas. In the formulas, *load_now* represents the current load, *nthreads* represents the number of runable threads, *ncpus* specifies the number of CPUs, and *sched_load* represents the load average.

$$load_now = \frac{nthreads}{ncpus}$$

$$sched_load = \frac{sched_load + load_now}{2}$$

If the number of threads is less than the number of CPUs, *load_now* is set to 1.

Both of the load-average calculations include a division operation; the OSF/1 scheduler avoids using floating-point division operations by scaling the *nthreads* value by a large factor and by shifting bits to the left to perform division by 2. The scaling factor is removed when a thread's CPU utilization is converted to its scheduling priority (see Section 5.1.2.4). There are no floating-point operations within the scheduler or in the rest of the kernel, because floating-point operations are expensive on some architectures.

5.1.2.3 The Calculation and Aging of CPU Utilization

As mentioned before, the scheduler increments a thread's CPU utilization during each clock interrupt. The scheduler increments the utilization according to the following formula:

$$sched_usage = sched_usage + (\Delta usage \times sched_load)$$

Once a second, each thread's utilization is aged according to the following formula (Δt is the number of seconds that have elapsed):

$$sched_usage = \left(\frac{5}{8}\right)^{\Delta t} sched_usage$$

The factor of 5/8 was chosen because it can be implemented by shifting and adding bits, and it produces good scheduling behavior.

5.1.2.4 Converting CPU Utilization to Priority

The conversion of a thread's *sched_usage* to a scheduling priority is implemented as a bit-shifting operation. The thread's priority is in the six leftmost bits of *sched_usage*, and the scheduler shifts bits to the right to determine the thread's priority. The shift operation removes the scaling factor introduced by the load average calculation and converts *sched_usage*

into a priority number between 0 (zero) and 31. The lower a thread's priority number, the better the thread's priority.

5.1.2.5 When Priorities Are Updated

The OSF/1 scheduler distributes the overhead of priority calculation by making each thread responsible for updating its own priority. The kernel maintains a global variable, *sched_tick*, which is updated once a second. The kernel also maintains a timestamp for each thread (the *sched_stamp* field within the thread data structure). The scheduler updates a thread's timestamp each time it updates the thread's priority. When the clock interrupts a thread's execution, the clock handler checks the value *sched_tick* against the thread's timestamp. If one or more seconds have passed since the thread's timestamp was updated, the clock handler calls the kernel's **update_priority( )** routine to update the thread's priority.

Distributing priority updates in this manner requires that each thread execute in order to update its priority. There are, however, instances in which threads with low priorities are unable to update their priorities because other threads with better priorities are monopolizing the CPU. Consequently, once every two seconds, the scheduler scans the run queues and updates threads that have been unable to update themselves.

Threads that are blocked cannot update their priorities and are not on a run queue; therefore, they cannot be updated when the scheduler scans the run queues. They must defer priority updating until they become runable again. When the kernel makes a thread runable, it checks the thread's timestamp against *sched_tick* and updates the thread's priority, if appropriate, before placing the thread in the run queue.

5.1.3 The Run Queue Data Structure

The kernel implements run queues using the run queue data structure. This structure contains the following elements:

runq[ ] The array containing the actual queues. This array contains 32 queues, numbered 0 (zero) through 31. These queues are doubly linked lists of threads. The array of queues is called

the *run queue*; each queue within the array is called a *sched queue*.

low A hint that specifies the run queue number that may contain the thread with the highest scheduling priority. The kernel maintains this hint to optimize the search for the thread with the best priority. The kernel cannot guarantee that the thread with the best priority is in this queue, but it can guarantee that the thread is not in a queue higher than this.

count The number of runable threads currently placed in the run queue. This field also optimizes the search for runable threads. If the count is 0 (zero), the kernel does not scan the run queue for a runable thread.

A thread's current scheduling priority is maintained in the thread data structure's *sched_pri* field. The value in this field maps directly to a sched queue within the thread's assigned run queue. For example, a new user thread that is runable and waiting to execute has a priority of 12, by default. The kernel schedules it for execution by placing it in the run queue's twelfth sched queue. In OSF/1, the lower the value of a thread's *sched_pri* field, the better the thread's priority. A thread with a priority of 6 has a better priority than a thread with a priority of 12.

When the kernel places a thread on a run queue, it places it at the tail of the appropriate queue for that priority. When the kernel chooses a thread for execution, it removes a thread from the head of the priority's queue.

5.2 Thread Execution States

In addition to maintaining a thread's priority, the scheduler subsystem manages a thread's transition between various states of execution. For example, a thread may be running, or runable and in a run queue, or it may be waiting for an event such as the release of a lock or the paging in of data.

The thread data structure includes a *state* field, which specifies a thread's current execution state. There are four basic state values, which can be combined to form other state values:

TH_RUN The thread is either executing or on a run queue, ready to be scheduled.

TH_WAIT The thread is waiting for a system resource to become available. For example, the thread may be waiting for a lock to be released or for data to be paged in.

TH_SUSP The kernel has asked the thread to stop executing. There are two major reasons for the kernel to suspend a thread: the thread is about to be terminated, or the thread's task is about to be swapped out. See Chapter 7 for a discussion of task swapping.

TH_SWAPPED
The thread's kernel stack has been *unwired*, and so its contents may not be in resident memory. Note that this state pertains only to the thread's kernel stack. The state *does not* indicate that the thread's task has been swapped out.

5.2.1 The Suspend Mechanism

The kernel suspends a thread through the **thread_hold()** and **thread_block()** routines; **thread_hold()** changes the thread's *state* field to indicate a suspend is pending, and **thread_block()** suspends the thread. The kernel uses the **thread_release()** call to resume a suspended thread's execution. The **thread_hold()** and **thread_release()** routines are not available to users. Users can suspend and resume threads with the **thread_suspend()** and **thread_resume()** routines.

At a given time, there may be multiple reasons for a thread to be suspended. Each thread data structure includes a *suspend_count* field that the kernel increments when it suspends the thread. If a thread's suspend count is non-zero when **thread_hold()** is called, the thread is already in a suspended state; **thread_hold()** merely increments the count.

The **thread_release()** routine decrements a thread's suspend count and releases the thread from its state of suspension if it is appropriate to do so. If the suspend count is greater than 1 at the time of the call, **thread_release()** merely decrements the thread's suspend count. If the thread's suspend count is 1 at the time of the call, **thread_release()** sets the suspend count to 0 (zero) and releases the thread from its suspended state. What happens to the thread next depends upon other aspects of the thread's execution state. If the thread is runable (it does not have a pending wait and its kernel stack is not swapped out), **thread_release()** dispatches the thread

to an idle processor if one is available; otherwise, the thread is placed on the run queue.

In addition to maintaining an internal suspend count, each thread also maintains a separate user suspend count so that user suspend operations (via **thread_suspend()** and **thread_resume()**) do not interfere with kernel suspend operations. When the user suspend count is incremented to 1, the kernel increments the internal suspend count. Subsequent increments of the user suspend count do not affect the internal suspend count. When the user suspend count is decremented from 1 to 0 (zero), the kernel decrements the internal suspend count by 1.

5.2.2 Execution State and the Suspend Mechanism

The diagram in Figure 5-1 shows the execution state transitions a thread may pass through with the suspend mechanism. The figure uses abbreviations to represent the following states:

Rq The thread is ready to run and is waiting on the run queue.

RqS The thread is on the run queue and a suspend state is pending.

R The thread is running on a processor.

RS The thread is running on a processor and a suspend state is pending.

S The thread is suspended.

SO The thread is suspended and its kernel stack is swapped out.

RO The thread is runable as soon as its kernel stack is swapped in.

RSO The thread is runable as soon as its kernel stack is swapped in, and it is resumed. This happens if a thread is suspended while the kernel stack is being swapped in.

The figure also uses abbreviations to indicate which kernel routines perform the state transitions:

choose Choose a new thread for execution. This is an abbreviation for the kernel's **choose_thread()** routine.

block Block the current thread's execution and context switch to a new thread if possible. This is an abbreviation for the kernel's **thread_block( )** routine.

hold Indicate that a transition to a suspended state is pending. This is an abbreviation for the kernel's **thread_hold( )** routine.

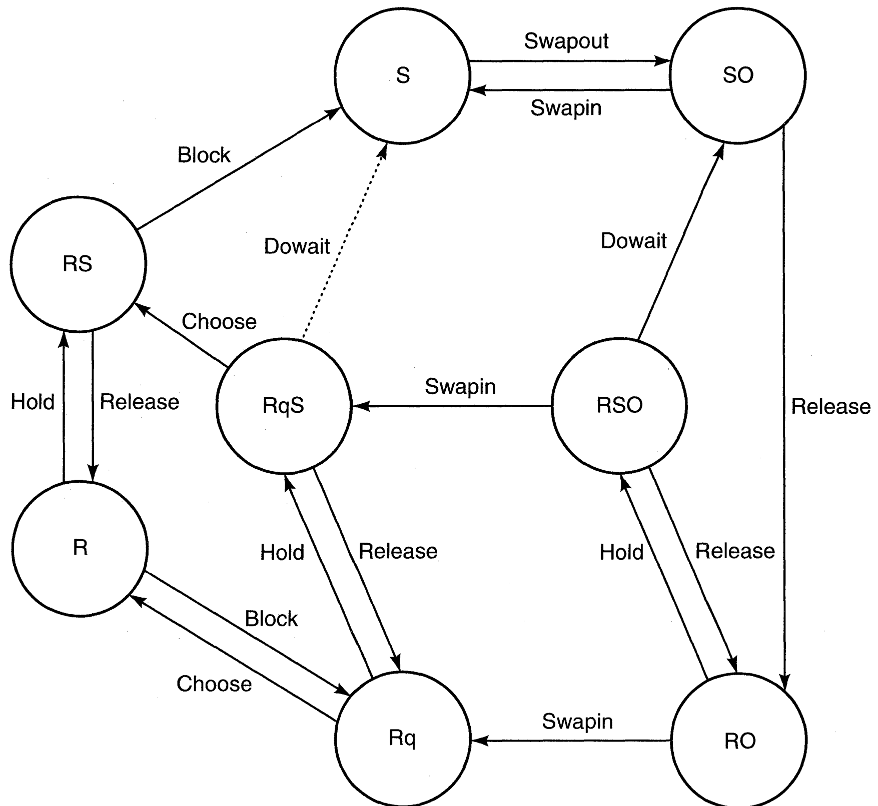
release Release the thread from a suspended state or pending suspended state. This is an abbreviation for the kernel's **thread_release( )** routine.

dowait Wait for a pending suspension to take place. This is an abbreviation for the kernel's **thread_dowait( )** routine. This routine is usually called by a thread that is waiting for another thread to become suspended.

swapout Swap out the thread's kernel stack. This is an abbreviation for the kernel's **thread_swapout( )** routine.

swapin Swap in a thread's kernel stack. This is an abbreviation for the kernel's **thread_swapin( )** routine.

Figure 5–1. Suspend Mechanism State Diagram



As shown in the figure, if a thread on the run queue has a suspend state pending, it transitions into the suspended state in either of two ways:

- The kernel chooses it for execution and executes the thread until it calls **thread_block**( ), thus becoming suspended.
- Another thread forces the thread to suspend by calling the **thread_dowait**( ) routine; this routine removes the thread from the run queue and leaves it in a suspended state.

Note that this latter transition (the transition between **RqS** and **S**) is represented by a dotted line. If a thread is not interruptible, the kernel does not allow this transition to occur; a pending suspend cannot take effect until after the thread runs on a processor.

5.2.3 The Event-Wait Mechanism

Threads often attempt to access system resources that the kernel cannot immediately supply. For example, a thread may need to lock a data structure that is currently locked by another thread, or it may need to access data that is not in main memory and so must be copied in from secondary storage. In circumstances such as these, the thread must wait until the resource becomes available before it can continue execution. The kernel's event-wait mechanism allows a thread to sleep while the resource remains unavailable, and then wakes up the thread when the resource becomes free. The event-wait mechanism is always invoked by the thread itself; a thread cannot force another thread to wait for an event. A thread can wait for only one event at a time.

Sometimes threads attempt to wait for events that may never take place. To prevent such threads from waiting forever, the event-wait mechanism allows threads to sleep *interruptibly*, or to set wakeup timeouts. A thread that sleeps interruptibly can be woken up by a UNIX signal. While a thread sleeps interruptibly, the kernel can swap out the thread's kernel stack and it can suspend the thread. If a thread sets a timeout before going to sleep, the kernel will wake the thread when the timeout expires if the event has not occurred.

A thread that is issuing a request to wait must identify to the kernel the event it is waiting for. Events are specified by integers. If, for example, a thread is waiting for the release of a lock, it may use that lock's address in memory to specify the event.

The kernel places all waiting threads in the scheduler's *wait queue*. This queue is implemented as a hash table to optimize the lookup operation that occurs when the kernel wakes a thread. The hash table is an array of queues indexed according to the table's hash function. When a thread issues a wait request, the kernel derives an index into the array by running the hash function on the integer representing the event being waited for. The kernel then places the thread in the queue that exists at the derived index. The thread is chained to the queue through the thread data structure's *runq* field (a thread cannot be in a run queue and waiting for an event at the same time).

5.2.3.1 Invoking the Event-Wait Mechanism

Typically, a thread invokes the event-wait mechanism by calling the **assert_wait( )** routine to indicate that it is about to wait, and then calling the **thread_block( )** routine to yield the processor and begin the wait. The call to **assert_wait( )** places the thread on the wait queue, and the call to **thread_block( )** puts the thread to sleep.

Because there is an interval between when a thread places itself on the wait queue and when it actually goes to sleep, it sometimes happens that the event a thread is waiting for occurs before the call to **thread_block( )**. When an event happens before the thread goes to sleep, the kernel removes the thread from the wait queue. When the thread calls **thread_block( )**, that routine, instead of putting the thread to sleep, places it in the appropriate run queue, and the thread is eligible for execution.

In the interval between calling **assert_wait( )** and **thread_block( )**, the thread may clean up its state within the kernel before it goes to sleep. For example, the thread may release any locks it is holding. If the thread wants to be woken if the event does not happen within a certain timeframe, it sets the timeout before calling **thread_block( )**.

In some instances a thread needs to clean up state *after* the event it is waiting on has occurred. For example, a device driver thread may need to restart its device after a completion event occurs. A thread of this type can ensure its ability to properly clean up state by waiting in an uninterruptible state. A thread that waits in this manner may wait forever if its event does not occur.

If a thread calls **thread_block( )** without first calling **assert_wait( )**, the scheduler initiates a context-switch operation, and the calling thread is placed in a run queue.

5.2.3.2 Waking a Sleeping Thread

The kernel wakes a sleeping thread either because the event being waited for has occurred, or because a condition arises that requires the thread's

sleep to be interrupted. The event-wait mechanism implements the wakeup operation with the following interfaces:

thread_wakeup( )
: Wakes up all threads that are waiting for the event.

thread_wakeup_with_result( )
: Wakes up all threads waiting for the event and indicates the reason for the wakeup.

thread_wakeup_one( )
: Wakes up the first thread on the wait queue that is waiting for that event. Other threads that may be waiting for the event remain asleep.

clear_wait( )
: Wakes up the specified thread, either because the event has occurred or because the thread's sleep should be interrupted. Use of this call requires knowing the identity of the sleeping thread.

The **thread_wakeup** routines are actually macros that invoke the kernel's **thread_wakeup_prim( )** routine. This routine finds the appropriate wait queue by executing the wait hash function on the event. For each thread that is to be woken up, **thread_wakeup_prim( )** removes the thread from the wait queue. What happens to the thread depends on the thread's current execution state:

- If the thread's state indicates that it was just waiting, it is dispatched to an idle processor if one is available, or it is placed in a run queue.
- If the thread was sleeping interruptibly and a suspend is pending, the routine allows the thread to go into a suspended state.
- If the thread was sleeping uninterruptibly and a suspend is pending, the routine ignores the request for suspension and schedules the thread for execution.
- If the thread's kernel stack has been swapped out while the thread was asleep *and* no suspend is pending, **thread_wakeup_prim( )** sets the thread's state to indicate it is runable, and then inititates the swapin operation by calling **thread_doswapin( )**. When this routine completes, the thread will either have been dispatched to an idle processor or it will have been placed in a run queue.

5.2.4 Execution State and the Event-Wait Mechanism

The diagram in Figure 5-2 shows the execution state transitions a thread may pass through with the event-wait mechanism. The diagram also shows the state transitions that occur when the event-wait, the suspend, and the thread swap mechanisms interact. In addition to the abbreviations used in the previous state diagram (Figure 5-1), this diagram also uses the following state abbreviations:

RW The thread is executing on a processor and a wait is pending.

RS The thread is executing and a suspend is pending.

RqS The thread is in a run queue and a suspend is pending.

RWS The thread is executing and a wait and suspend are pending.

W The thread is in a wait queue.

WS The thread is in a wait queue and a suspend is pending.

WO The thread is in a wait queue and its kernel stack is swapped.

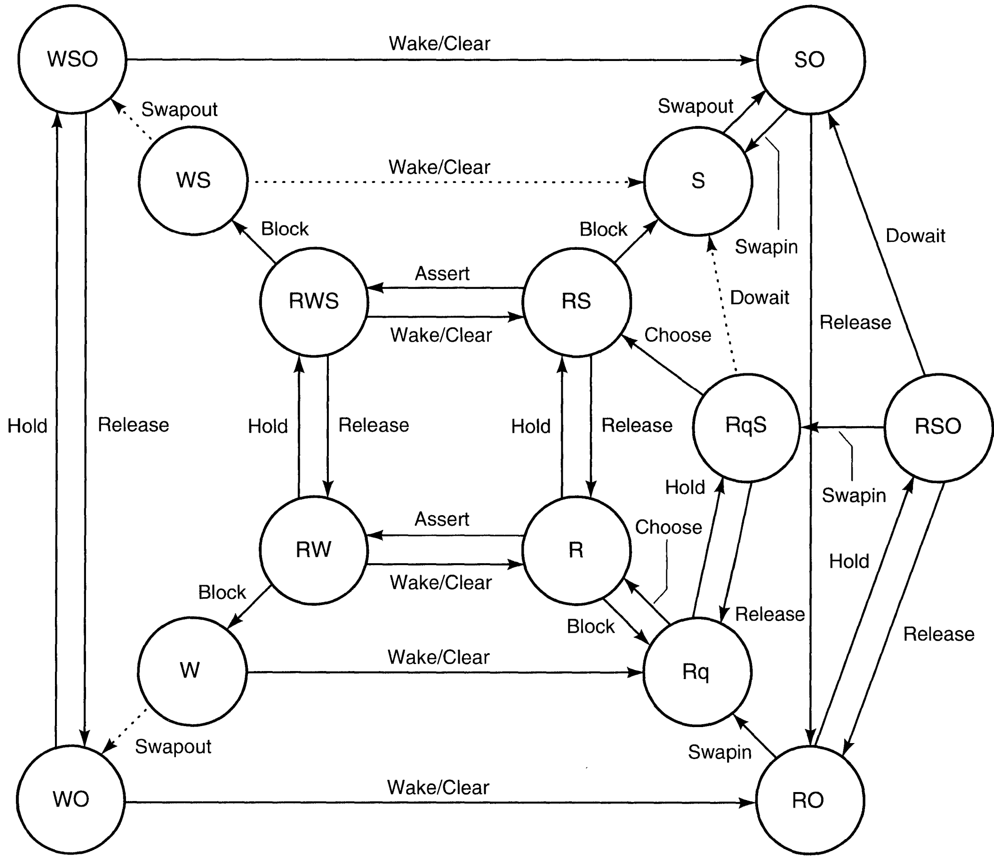
WSO The thread is in a wait queue, its kernel stack is swapped, and a suspend is pending.

Figure 5-2 also uses abbreviations to indicate which kernel routines perform the state transitions. In addition to the abbreviations used in Figure 5-1, this diagram includes the following:

assert Asserts that the thread is about to transition to a wait state. This is an abbreviation for the kernel's **assert_wait( )** routine.

wake/clear Removes the thread from the wait queue. This abbreviation stands for the **thread_wakeup( )** and **clear_wait( )** routines, respectively.

Figure 5–2. Event-Wait Mechanism State Diagram



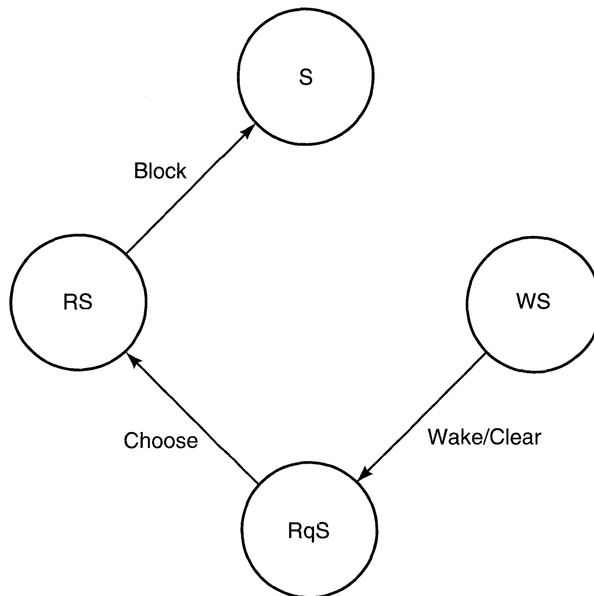
In this diagram, dotted lines indicate transitions that are disabled when a thread is not interruptible. These include the transitions between the following states:

- **WS** to **S**
- **WS** to **WSO**
- **W** to **WO**

Figure 5-2 does not show what happens to a thread that is waiting uninterruptibly with a suspend pending (**WS**) when it is awakened. As indicated in the figure, the transition to **S** is disabled when a thread sleeps

uninterruptibly. When the kernel wakes up such a thread, the kernel ignores the pending suspension and makes the thread immediately runable. If an idle processor is available, the kernel dispatches the thread to it. If a processor is not available, the kernel places the thread in the run queue. Although the thread is then running or runable, its pending suspend has not been cancelled. It will take effect the next time the thread calls **thread_block( )**. Figure 5-3 shows this series of state transitions.

Figure 5–3. State Transition of a Thread in an Uninterruptible Sleep



5.3 Scheduler Support for Parallel Applications

In addition to supporting timesharing applications, the OSF/1 scheduler supports the scheduling requirements of different parallel programming models. The scheduler's support for parallel programming models is furnished through two mechanisms: *processor sets* and *scheduling hints*.

The processor set mechanism allows the kernel to furnish an application with a dedicated set of CPUs; the application's threads execute on these CPUs without having to compete with other threads. The scheduling hints

mechanism allows an application to manage the scheduling of its threads by providing hints to the scheduler about the order in which to execute the threads.

5.3.1 Processors and Processor Sets

A *processor* is an object that represents a CPU. The OSF/1 kernel maintains a **processor** data structure for each of the system's CPUs and manages each CPU through this data structure. A processor set is an object that represents a set of processors. The kernel manages each processor set through its **processor set** data structure.

The system initialization procedure includes the creation of the kernel's *default processor set.* This processor set always exists because the kernel assigns its own threads to it. By default, the OSF/1 scheduler schedules threads on a timesharing basis; in this environment, all threads execute on the default processor set, and the kernel never creates additional processor sets. A parallel program can execute using processors from the default set, but cannot have processors dedicated to its execution.

The OSF/1 kernel interface includes a set of primitives that can be used to create and manage new processor sets. These interfaces do not, in themselves, provide for any sort of processor allocation policy. A processor allocation policy must deal with the following issues:

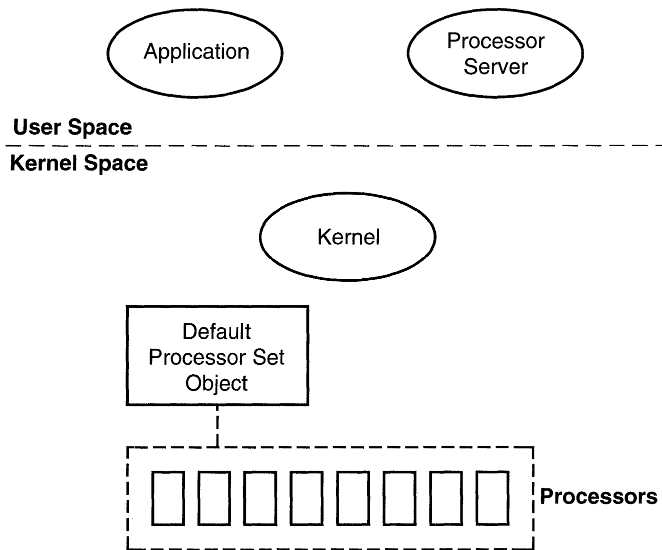
- Which applications are allowed to use dedicated processors
- Prioritization of applications competing for processors
- Amount of time an application is allowed to use dedicated processors
- Maximum number of processors an application is allowed to use on a dedicated basis

This type of policy is likely to vary from system to system; if this policy were coded into the kernel, the kernel would have to be modified and rebuilt each time the policy needed to be changed. In the processor set model, the processor allocation policy is handled by a user-level *processor server program.* A processor server program implements a given processor allocation policy and uses the kernel interfaces to affect the policy. Because it runs at user level, such a program can be reconfigured or replaced by another program without modification to the kernel. Processor server programs must run with privileges.

OSF/1 does not include a processor server program; however, the kernel interface includes a set of primitives that can be used to implement processor servers. The availability of the processor allocation primitives is configuration-dependent.

Figures 5-4 to 5-6 illustrate how a parallel application might interact with the kernel and a processor server program to acquire a set of processors dedicated for its use. The figures are necessarily schematic, and the description that accompanies them is a simplified one. In this example, the system hardware includes eight CPUs, which the kernel manages through eight processor objects. In Figure 5-4, all eight processors belong to the system's default processor set object. All of the system's active threads are assigned to this processor set.

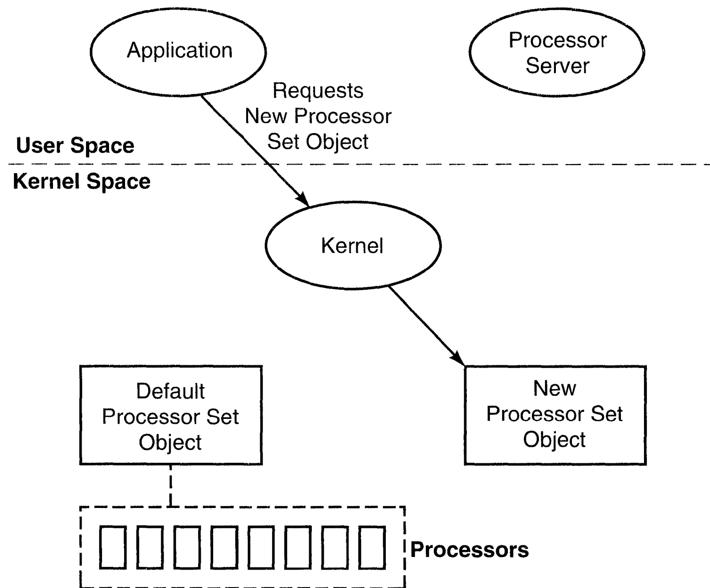
Figure 5–4. The Default Processor Set



In Figure 5-5, the parallel application asks the kernel to create a new processor set object, and the kernel does so. The new processor set does not initially have any processors assigned to it. An application does not need to

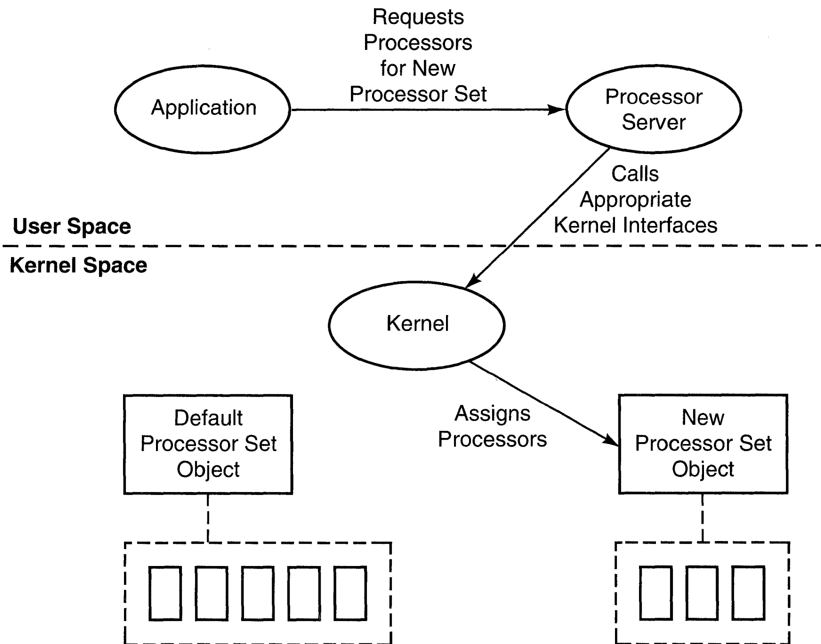
be privileged to create a new processor set, because all new processor setsare empty when created. Only the privileged processor server can ask the kernel to assign processors to a processor set.

Figure 5–5. An Application Allocates a Processor Set



In Figure 5-6, the application asks the processor server to assign processors to the new processor set. The server analyzes the request based on its policy. If it decides to honor the request, the server calls the appropriate kernel interfaces. The kernel responds to the server's request by reassigning processors from the default processor set to the new set. Now the application can assign its threads to the new processor set, and the scheduling system will force them to execute only on the processors of this set. Those threads that were already assigned to the default processor set will continue to run only on those processors that remain in that set, and any newly created threads that do not belong to the application will be assigned to the default set.

Figure 5–6. The Application Requests Processors; the Kernel Assigns Processors



Processor sets can be used to support various types of parallel applications. A *fine-grained* application, whose threads synchronize with one another very frequently, may require access to a number of CPUs that matches the number of threads in the application, while a *coarse-grained* application may be able to execute on a processor set that has fewer CPUs than the application has threads.

5.3.2 Scheduling Hints

Certain coarse-grained applications may be able to optimize their performance by being able to advise the scheduler how to schedule their threads. The OSF/1 kernel provides the **thread_switch()** interface, which allows threads to influence the scheduler's activities. When a thread blocks to synchronize with another, it can use **thread_switch()** to provide the kernel with a hint about which thread should be executed next.

Scheduling hints are as follows:

Mild Discouragement
: Switch contexts to another thread with an equal or higher priority to execute. If such a thread does not exist, continue executing the current thread.

Strong Discouragement
: Depress the current thread's priority and switch context to another thread. Block the current thread's execution and depress its priority until it executes again or until a specified timeout has expired.

Absolute Discouragement
: Stop the current thread's execution for a specifed period of time.

Handoff
: Bypass the scheduler entirely and switch contexts to the specified thread. (If both the current thread and the specified thread are time-sharing threads, the specified thread gets the remainder of the current thread's time-slice.)

The discouragement hints can be used to optimize the synchronization of threads that synchronize using *test-and-set* locks. These kinds of locks do not record the identity of the current lock holder; consequently, a thread waiting for the lock may not be able to identify the thread it is waiting on, and therefore cannot provide a handoff hint to the scheduler. The handoff mechanism can be used if a thread can identify the lock holder either because of the structure of the application or because the threads synchronize using locks based on compare-and-swap instructions.

Mild hints may not be useful in instances where an application executes in a timesharing mode, because multiple threads may simultaneously yield their processors to each other instead of to the thread or threads that hold the

synchronization locks. Absolute hints may be appropriate under these circumstances, but since their granularity is based on the frequency of the hardware's clock, they are not appropriate for medium-grained to fine-grained synchronization.

5.3.2.1 The thread_switch( ) Routine

The **thread_switch( )** routine encapsulates the scheduler's hint facility. When it is called to provide a mild discouragement hint, the routine simply calls **thread_block( )** to yield the processor to the thread with the highest priority.

A thread that calls **thread_switch( )** to provide an absolute discouragement hint must specify a timeout value with the call. In this instance, **thread_switch( )** places a call to the **thread_timeout( )** routine in the kernel's callout table. When the timeout expires, **thread_timeout( )** reschedules the thread for execution by calling the **clear_wait( )** routine.

A thread that calls **thread_switch( )** to provide a strong discouragement hint must also specify a timeout. In this instance, **thread_switch( )** depresses the thread's priority to the lowest possible value. The scheduler will restore the thread's previous priority when the thread next executes or when the supplied timeout has expired, whichever comes first.

A thread that wants to hand off the CPU to a specific thread identifies the thread with an argument to **thread_switch( )**. The routine finds the specified thread in the processor set's run queue, removes it, and switches context to it using a call to the **thread_run( )** routine.

5.3.2.2 The thread_depress_priority( ) Routine

The kernel implements strong discouragement hints with the **thread_depress_priority( )** routine, which can be used to temporarily depress a thread's priority. This routine manipulates the thread data structure's *depress_priority* field. Under normal circumstances, this field is set to -1. **thread_depress_priority( )** depresses a thread's priority by

moving the thread's base priority, (maintained in the thread's *priority* field) to the *depress_priority* field. The routine then sets the thread's *priority* and *sched_priority* to 31, the lowest possible priority value.

The kernel reverses this operation by restoring the thread's *priority* field from the *depress_priority* field. The kernel then resets the *depress_priority* field to -1 and recomputes the thread's scheduling priority.

5.4 CPU-Usage Timer Support

Traditional UNIX systems measure the amount of time a given process utilizes the CPU with **statistical timers.** Statistical timers are based on a time-sampling mechanism that is driven by the hardware's clock. When the clock interrupts an executing process, the clock interrupt handler charges the process with a full tick of CPU utilization. Charging CPU usage on a time-sampling basis introduces a margin of error, because the system cannot accurately determine exactly what happens in the timeframe between samples. Sampling errors include the following:

- A process that executes for a fraction of the timeframe between clock interrupts accrues a full tick's worth of utilization if it is executing at the interrupt time.
- Conversely, a process that executes for a fraction of the timeframe but is not executing when the clock interrupts accrues no utilization.
- The kernel may have been executing interrupt code (on behalf of another process) during the timeframe but utilization is still charged to the current process.

The margin of error introduced by a time sampling mechanism varies depending upon the length of a process's execution. A long-running process is more likely to accrue time sampling errors both for and against it; if a process runs for a sufficient amount of time, these discrepancies tend to cancel each other out. However, discrepancies associated with short-running processes may be significant.

Although timesharing systems measure a process's CPU utilization to determine the process's scheduling priority, statistical timing discrepancies are generally not large enough, even for short-lived processes, to adversely affect the scheduling algorithm. However, other timing applications, such as

those that profile a program's execution, can be severely affected by such discrepancies.

Certain hardware platforms provide support for the implementation of *CPU-usage timers*, which measure CPU utilization directly instead of statistically. These timers can improve the accuracy of statistical measurements by as much as three orders of magnitude.

CPU-usage timers are based on a *timestamp* mechanism. The hardware maintains a timestamp source that is incremented at a known rate. A timer can measure the length of time that has elapsed during a given activity by reading the timestamp source at the beginning of the activity, reading it again at the end of the activity, and taking the difference between the two readings.

OSF/1 provides software-implemented interval timers that can be used on hardware platforms that provide a source of timestamps. The scheduler uses these timers to measure a thread's CPU utilization. They are also available to profilers and other applications that need to measure time accurately. On platforms that do not provide interval timer support, the OSF/1 scheduler uses a statistical time sampling mechanism to determine a thread's utilization of CPU resources.

5.4.1 OSF/1 Timers

The kernel measures each thread's consumption of CPU resources through the thread's *user timer* and *system timer*. The kernel uses the thread's user timer to measure the amount of time the thread executes in user mode, and it uses the system timer to measure the amount of time the thread executes in kernel mode. The scheduler determines a thread's total CPU utilization by adding together the measurements taken by both timers.

The kernel also maintains a separate interval timer, a *kernel timer*, for each of the system's CPUs. The kernel uses these timers to measure the amount of time the kernel spends executing interrupt code. Kernel timers ensure that the consumption of CPU resources during interrupt handling is not charged to the interrupted thread. Currently, a kernel timer accumulates time for every interrupt handled by its processor, regardless of the interrupt type. The timer facility can be extended with additional kernel timers to measure

interrupts based on type. The system can then be monitored to determine how much time is being devoted to the various interrupt sources and how that time is being distributed across the system's CPUs.

At any given moment, a CPU's usage is measured by either a thread's user timer, a thread's system timer, or the CPU's kernel timer. The kernel ensures that utilization is charged accurately by providing routines to switch between timers. Timers must be switched as follows:

- When a thread executing in user mode traps into the kernel through a page fault, exception, or system call, the kernel must switch from the thread's user timer to its system timer.
- When a thread returns to user mode from kernel mode, the kernel must switch from the thread's system timer to its user timer.
- When the kernel interrupts a thread's execution to handle an interrupt, the kernel must switch from the thread's current timer to the CPU's kernel timer.
- When the kernel returns control from an interrupt, it must reactivate the interrupted thread's current timer, either its user or system timer.
- When the kernel context switches between threads, it must switch from the blocking thread's system timer to the new thread's system timer.

5.4.2 The timer Data Structure

All OSF/1 timers are implemented by the kernel's **timer** data structure. This structure includes the following fields:

low_bits — Accumulated time in units corresponding to the units used by the hardware's timestamp source. For example, this field may accumulate time in microseconds.

high_bits — Accumulated time in *normalized* units. When the *low_bits* field is about to overflow, the kernel converts its value to a normal time value and adds it to the contents of the *high_bits* field. If, for example, *low_bits* accumulates time in microseconds, the *high_bits* field may accumulate time in seconds.

tstamp The value of the timestamp source when the current activity began.

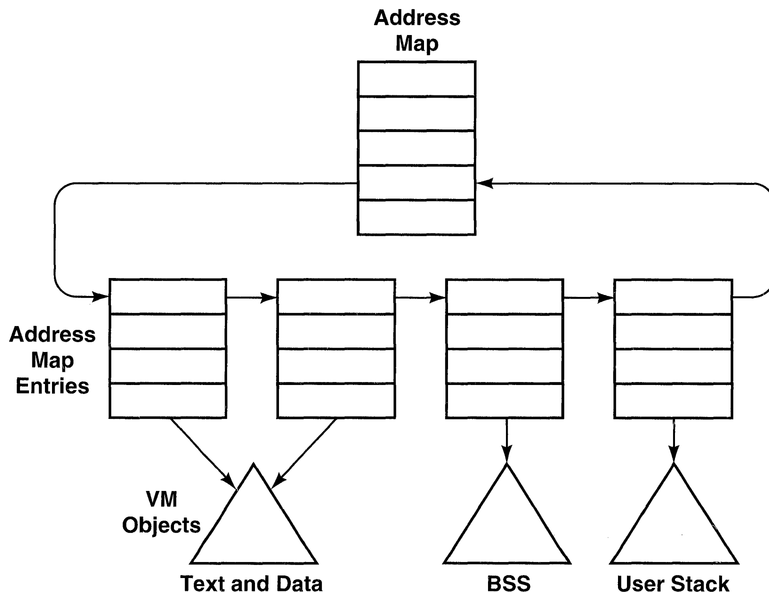
When the kernel activates a timer, it sets the timer's *tstamp* field to the value of the timestamp source. When the kernel is about to switch to another timer, it subtracts the *tstamp* value from the current value of the timestamp source and adds the result to the timer's *low_bits* field. The kernel then switches to the next timer.

Chapter 6

The Virtual Memory Subsystem: Address Space Implementation

Chapter 3 introduced the major features of the Virtual Memory subsystem (VM) and briefly described the data structures that implement virtual address spaces in Mach. To review, a task's address space is implemented through its *address map*. The address map contains *address map entries*, each of which maps a range of virtual addresses to a *Virtual Memory object* (VM object). See Figure 6-1.

Figure 6–1. Implementation of Task Address Space



6.1 Address Maps and Address Map Entries

A task's address map is represented by a **vm_map** structure; the map's address map entries are represented by **vm_map_entry** structures. As shown in Figure 6-2, the **vm_map_entry** structures are chained together in a doubly linked list, and the head and tail of this list are linked to the **vm_map** structure.

This arrangement of structures supports the fast lookup of virtual addresses during page fault handling. The structures also support the compact representation of large sparsely filled address spaces because

- The address map maintains address map entries only for allocated regions of address spaces.
- Each address map entry may map an arbitrarily large range of addresses.
- The address map entries are represented by small, fixed-size data structures.

6.1.1 The vm_map Data Structure

The **vm_map** structure includes the following fields:

min_offset Specifies the address map's first allocated virtual address. This field corresponds to the first address map entry's starting virtual address.

max_offset Specifies the map's final allocated address. This field corresponds to the last address map entry's ending address. The page fault handler uses this field and the *min_offset* field to quickly check the validity of a faulting address. If the address does not lie between *max_offset* and *min_offset*, it is invalid.

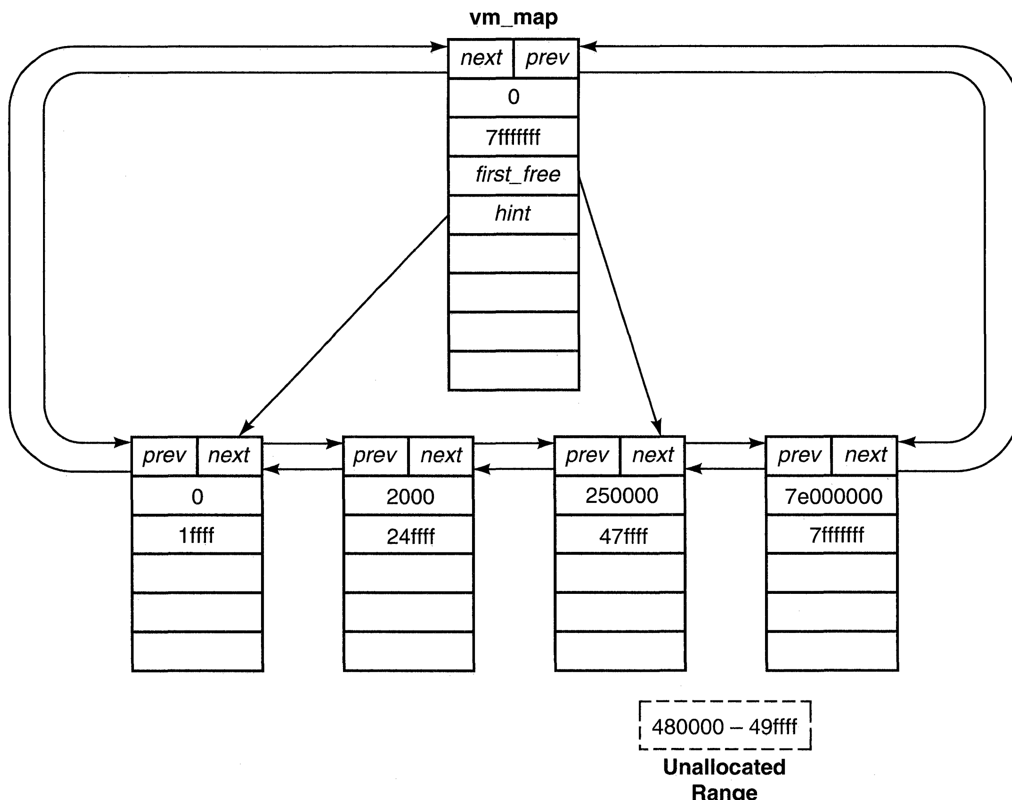
first_free Points to the first unallocated region within the task's address map. The kernel uses this hint when mapping data to arbitrary locations within the address space.

hint Used to optimize the page fault handler's lookup of virtual addresses. Usually the *hint* field points to the address map entry that maps the address that generated the most recent search operation. If the task is generating a series of page faults in the same region of memory—a condition that arises frequently—the hint optimizes virtual-to-physical translation. However, the field does not always point to the last successful lookup. When a task allocates a new region of memory, the kernel updates the field to point to the new region's address map entry.

The **vm_map** structure also contains a pointer to the process's physical map (pmap). The kernel uses this pointer to access the task's pmap when invoking operations on that object.

The address space described by the **vm_map** structure in Figure 6-2 contains four regions of virtual memory. The first three regions are contiguous to one another and span the range between 0x0 (zero) and 0x47ffff. The *first_free* field points to the maps third address map entry because the first range of unallocated address space lies beyond the range mapped by the address map entry (the dashed box indicates that the range of addresses between 0x480000 and 0x49ffffff is unallocated). The **vm_map**'s *hint* field points to the first address map entry, indicating that the last page fault was generated by a reference within the address range mapped by this entry.

Figure 6–2. A vm_map Structure and Its vm_map_entry Structures



6.1.2 The vm_map_entry Data Structure

Each **vm_map_entry** structure maps a range of virtual addresses to a set of pages in an object. The length of the range implies the number of pages that are mapped. For example, if the system's virtual page size is 0x1000 (4K bytes), a virtual range with a length of 0x1000 maps to a single page, a range with a length of 0x2000 maps to two pages, and so on. The **vm_map_entry** structure maps the address range to the object's pages with an *object/offset* reference. The object reference specifies the object that the address map entry maps to, and the offset value specifies the location within the object where the mapping begins. For example, if the offset value is

0x0, the mapping begins with the object's first page. If the offset value is 0x1000 (and the system's virtual page size is 0x1000), the mapping begins with the object's second page, and so on.

The **vm_map_entry** structure includes the following fields:

start The beginning of the virtual address range being mapped.

end The end of the range being mapped.

protection The protection at which the range is mapped. For example, a region can be mapped read-only, or read/write.

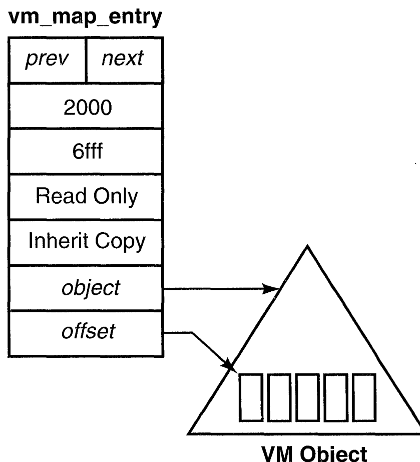
inheritance Specifies whether or not child tasks will inherit the mapped region, and if so, whether they will inherit a copy of the region or inherit shared access to the region.

object The object that is mapped to the address range.

offset The mapping's offset into the object.

Figure 6-3 shows the relationship between a **vm_map_entry** structure and the VM object it maps. The VM object is mapped to the address range that spans 0x2000 to 0x6fff. The object is mapped read-only and inherit-copy at an offset of 0x0. The size of the address map entry's virtual range (0x4fff) implies that the address map entry maps the object's first five pages.

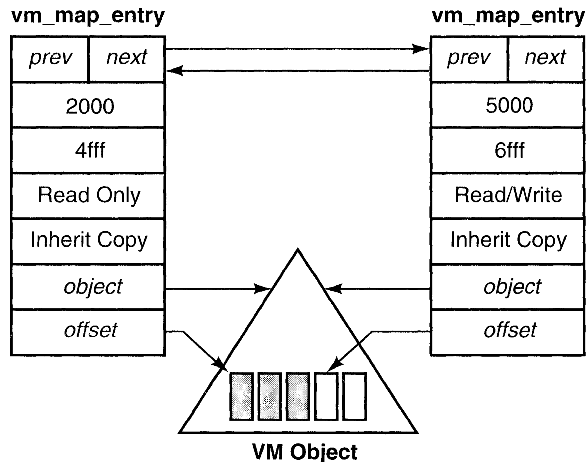
Figure 6–3. A vm_map_entry Structure and the VM Object It Maps



Usually a region of allocated memory is mapped through one address map entry, but this is not always so. A region may be mapped by multiple address entries. If all of a region's addresses have the same protection and inheritance attributes, the region can be represented by a single address map entry. If the region contains subsets of addresses that have different attributes, the region requires an address map entry for each subset.

Consider again the preceding example. The region described contains the virtual addresses between 0x2000 and 0x6fff. Suppose the task calls the **vm_protect**( ) interface to change the protection of the addresses between 0x5000 and 0x6fff from read-only to read/write. Figure 6-4 shows the result of this operation.

Figure 6–4. Changing Protection on a Range of Virtual Memory



The system splits the original address map entry into two separate entries. The first address map entry maps the virtual addresses between 0x2000 and 0x4fff to the VM object's first three pages; the address map entry references the VM object at an offset of 0x0, and the virtual range implies that the mapping has a length of 0x3000. This mapping retains the read-only protection. The second address map entry maps the 0x2000 range of addresses between 0x5000 and 0x6fff to the VM object's fourth page by referencing the VM object at an offset of 0x3000. The address map entry maps this data read/write.

6.1.3 Address Map Entries and the Page Fault Handler

When it invokes the page fault handler, the kernel analyzes the fault, and then passes the handler the virtual address whose reference generated the fault and a pointer to the faulting task's address map. If the virtual address is valid, the faulting address lies in one of the map's ranges of allocated virtual memory. Before the handler can resolve the fault, it has to locate the address map entry that represents this range. This address map entry references the VM object that manages the faulting address's virtual page.

When it has located the address map entry, the handler calculates the faulting address's offset into the region, then uses that offset to identify the virtual page that contains the faulting address's physical counterpart. The handler can then determine the physical location of the virtual page and thereby resolve the page fault.

Consider Figure 6-3. Suppose in this example that a task has generated a page fault trying to reference an address between 0x5000 and 0x5fff. The page fault handler proceeds as follows (assume a system page size of 0x1000):

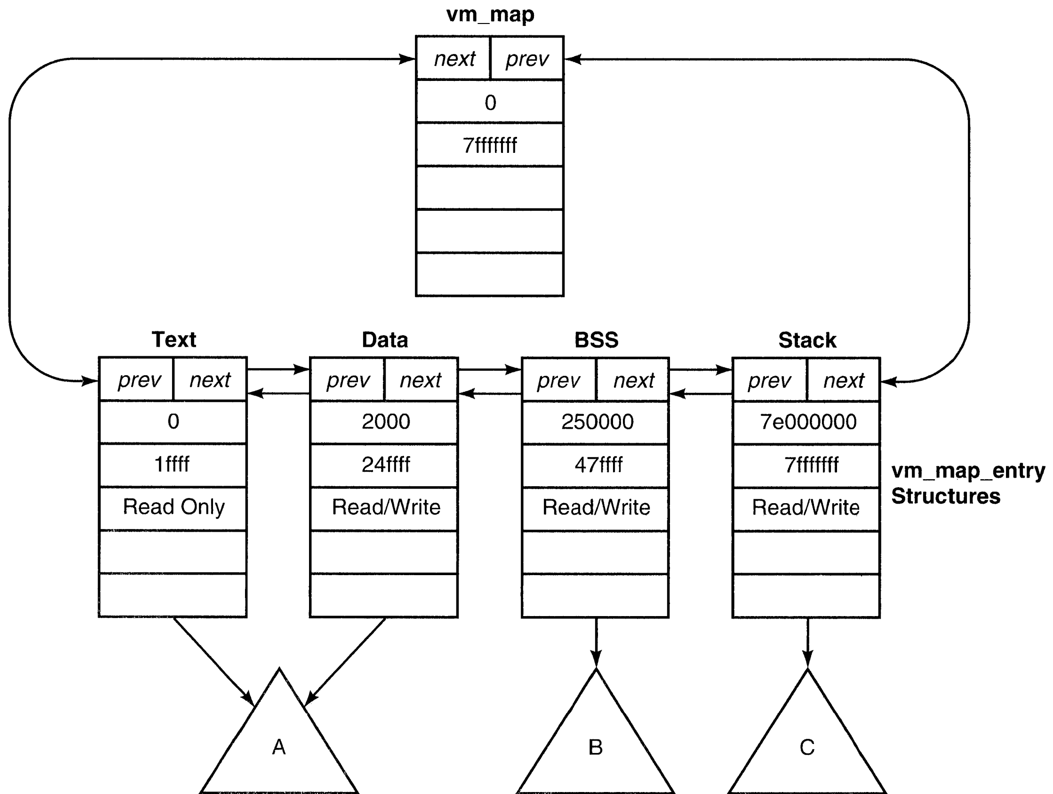
1. The handler locates the faulting address's address map entry.
2. It calculates the address's offset into the region. In this example, the faulting address lies between 0x5000 and 0x5fff, so the address's offset is between 0x3000 and 0x3fff into the region.
3. The handler adds this offset to the VM object's offset to determine the virtual page's offset in the object. In this example, the VM object is mapped at an offset of 0x0, so the virtual page in question has an offset of 0x3000 into the object. It is in the object's fourth page.
4. The handler now determines if the page is resident by using the object/offset value to search the resident page table. If the page is not resident, it must be paged in from secondary storage.

6.2 Implementation of UNIX Process Address Spaces

Figure 6-5 shows how OSF/1 implements a typical UNIX process's address space. As shown in the figure, the process's address map contains four address map entries. However, the process's address map maps three VM objects into the address space. VM object **A** represents the process's text and initialized data, which is collectively mapped from the process's executable file; VM object **B** represents the process's uninitialized data and heap; VM object **C** represents the process's user stack.

Note that two address map entries map to VM object **A**. The first address map entry maps the text portion of the VM object and the second address map entry maps the initialized data portion of the object. The sections are mapped with different protection attributes; the text is mapped read-only with execute, and the initialized data is mapped read/write.

Figure 6–5. Implementation of a UNIX Process Address Space



6.3 The Optimization of Virtual Copy

Tasks often inherit regions of virtual memory from their parents. Inherited regions are of two types: shared regions and copied regions. If a child shares an inherited region with its parent, it will see all modifications the parent makes to the mapped data, and the parent will see all modifications made by the child. If a child process inherits a copied region, it will not see changes made by the parent, and the parent will not see changes made by the child. Shared regions are discussed in Section 6.5. This section describes the implementation and optimization of copied regions of memory.

The VM system uses copy-on-write mechanisms to optimize the copying of virtual memory. Copy-on-write allows tasks to share mappings to the same data read-only, and data is copied only when one of the tasks attempts to write the data.

In OSF/1, copy-on-write operations happen on a page-by-page basis. For example, if two tasks share several pages copy-on-write, and one of the tasks wants to write data on the first page, the kernel will copy only that page. The other pages will remain uncopied until one of the tasks attempts to write them.

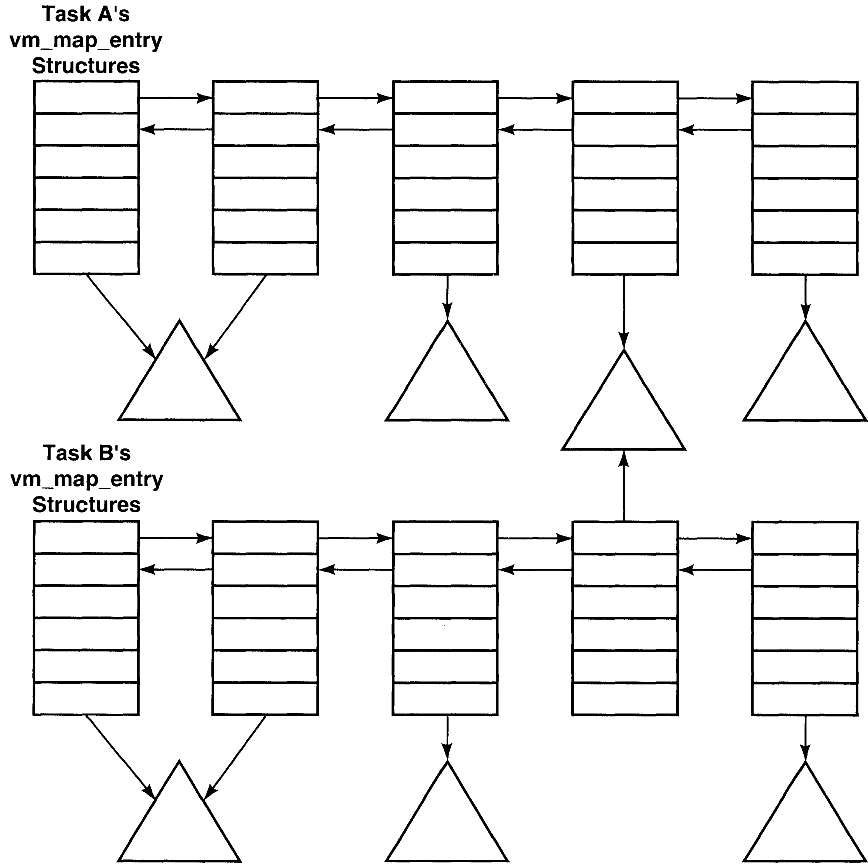
There are two types of copy-on-write operations: *symmetric* copy-on-write and *asymmetric* copy-on-write.

6.3.1 Symmetric Copy-on-Write

Symmetric copy-on-write is the kernel's mechanism for optimizing the virtual copy of temporary data. This is data that is created in memory as a task executes. For example, a UNIX process's heap contains temporary data.

In Figure 6-6, tasks A and B share a region of data copy-on-write because B has inherited from A a virtual copy of the region. As indicated in the figure, the system implements the virtual copy by mapping the data's VM object into both tasks' address spaces.

Figure 6–6. Two Tasks Share Data Copy-on-Write



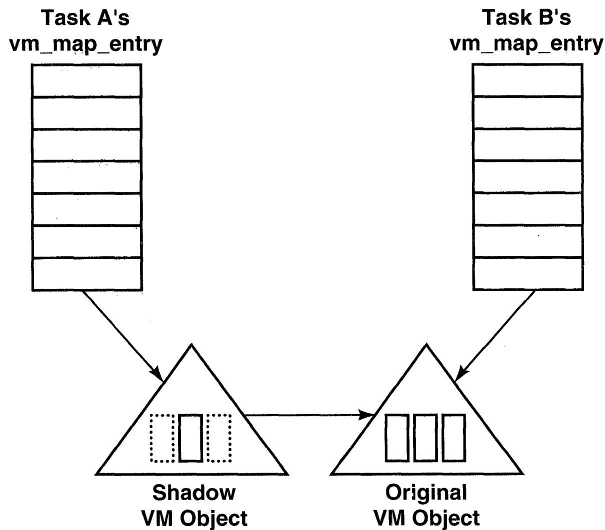
Suppose that task A needs to modify data mapped to its copy of the region, and the data to be modified is contained in the VM object's second page. The system cannot allow task A to modify the original page's data because the page is currently being shared with task B. At this point the system must

provide task A with an exclusive copy of the affected page. The system proceeds as follows:

1. It allocates a new physical page to hold the copied data and copies the data from the original page to the new page.
2. It initializes a new VM object, called a *shadow object*, and places the new page there.
3. It changes the mapping of task A's region from the original VM object to the shadow VM object.

Now task A has an exclusive copy of the original object's second page. Figure 6-7 shows the result of this copy-on-write operation.

Figure 6–7. Task A Writes Data

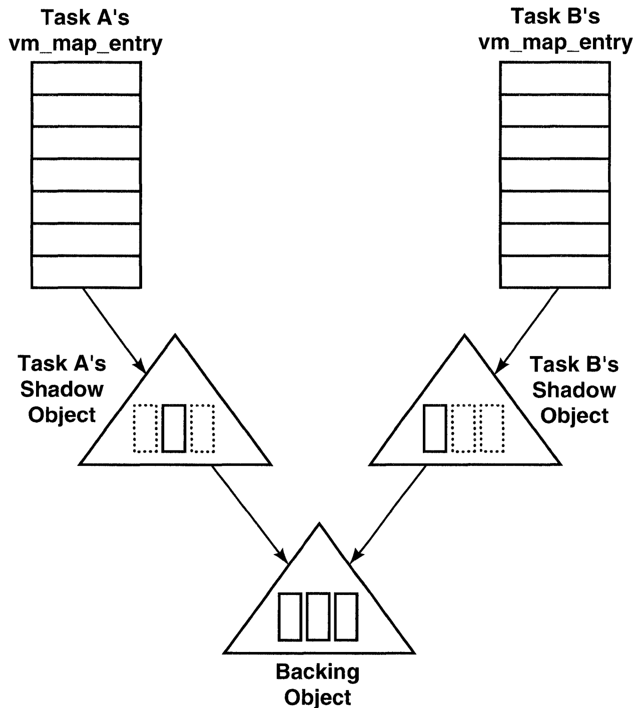


The shadow object is the means by which the system manages task A's changes to the shared data. Whenever task A needs to modify a previously unmodified page in the original VM object, the system allocates a new page, copies the required data from the original page, and inserts the new page in task A's shadow object. The task no longer references the original page; it references the page that resides in the shadow object.

The system moves pages to the shadow only when the task needs to modify data; unmodified pages remain in the original VM object. Suppose, for example, that task A wants to read (but not modify) data contained in the original VM object's third page, which has not been modified. The task first looks for the page in its shadow object. The task then moves to the original object, finds the page, and reads its data. The original object is referred to as the shadow object's *backing object*.

Now suppose that task B wants to modify data mapped to its region that is associated with the original object's first page. As shown in Figure 6-8, the system responds by creating a shadow object for task B. The system allocates a new physical page, copies the data from the original object's first page, and inserts the new page in task B's shadow object. The system maps the shadow to task B's region. The task then has an exclusive copy of the original object's first page.

Figure 6–8. Task B Writes Data



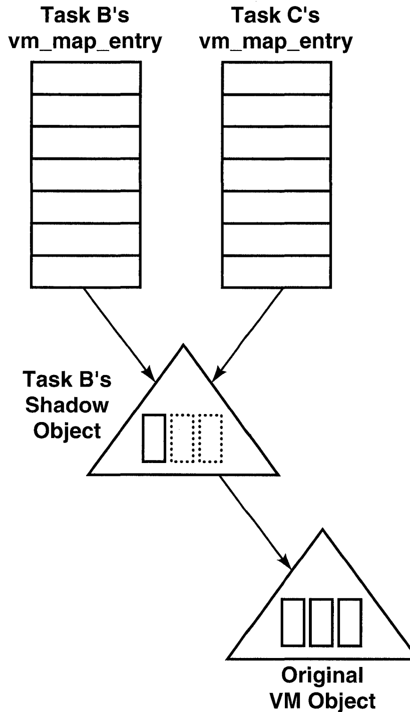
This version of the copy-on-write optimization is known as symmetric copy-on-write because the system must create a shadow object for each task when that task needs to modify the shared data. For reasons that will be explained in Section 6.3.2, symmetric copy-on-write is appropriate only for VM objects that manage temporary data. The virtual copy of permanent data can also be optimized in a copy-on-write fashion, but the mechanism cannot be implemented in a symmetric fashion.

6.3.1.1 Shadow Object Chains

Shadow VM objects can themselves be subject to copy-on-write. In such an instance, the shadow VM object becomes the original VM object, with its pages subject to copy-on-write.

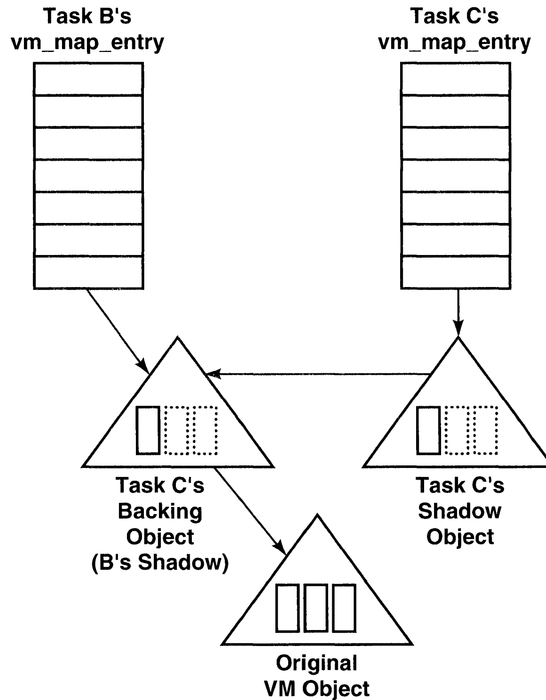
For example, in Figure 6-9, task A and task B share data copy-on-write, and task B has written to the original object's first page, thus generating a shadow object. Task B subsequently creates task C. Task C inherits from B a virtual copy of the data. This data is managed by two VM objects: B's shadow object, which manages the single page B has modified, and the original VM object, which manages the two pages B has yet to modify. Tasks B and C both map the shadow object copy-on-write.

Figure 6–9. Tasks B and C Share Data Copy-on-Write



Suppose that task C wants to write the first page of data. The system allocates a new physical page to copy the data to, and creates a new shadow VM object to manage the new page. (See Figure 6-10.) The system maps this object to task C's address space, and now the task is free to write the data. Note that the shadow object has as its backing object task B's shadow object, which in turn has as its backing object the original VM object. As before, if either task B or C wants to access data contained in the other pages, it gets the data from the original VM object. This collection of VM objects is known as a *shadow chain.*

Figure 6–10. Task C Writes Data



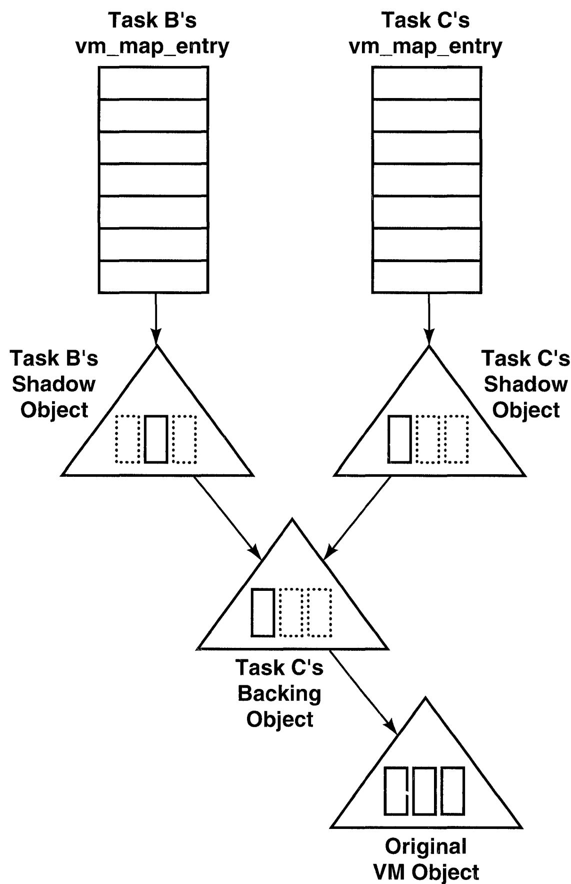
Suppose that task B wants to write the second page of data. The system proceeds with the copy-on-write operation as follows:

1. The system allocates a new page to hold the copied data.
2. The system follows the shadow chain references back to the original VM object, finds the data's page there, and copies the data to the newly allocated page.
3. The system creates a new shadow object to manage the new page and maps this object to task B's address space. B can then proceed to modify the page's data.

Figure 6-11 illustrates the results of this operation; task B's former shadow object becomes the current shadow's backing object. If task B needs to access data from the first page, it must follow the shadow chain back to its

former shadow object. If the task wants to access data from the third page, it must follow the shadow chain to the backing object and then on to the original VM object.

Figure 6–11. Task B Writes Data, Creating a Shadow Tree



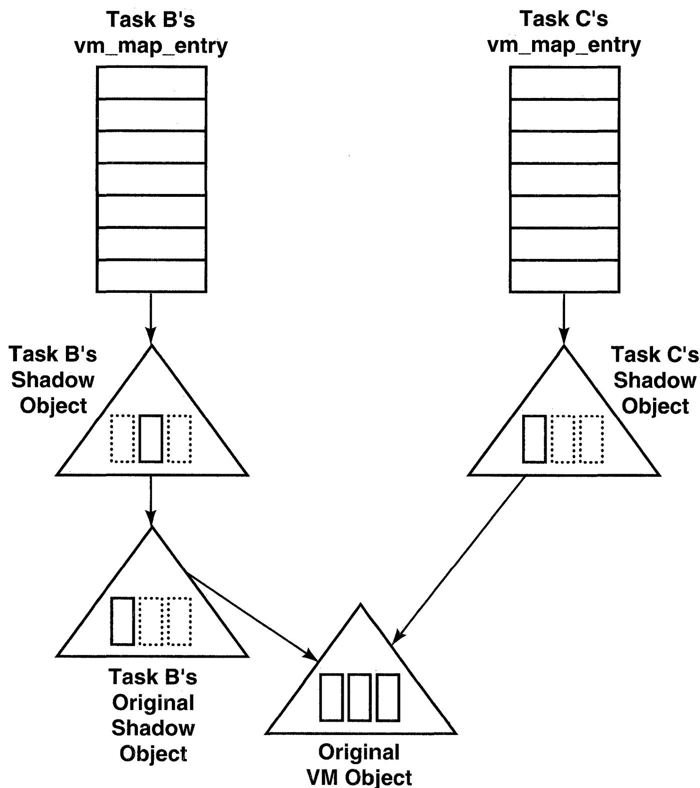
6.3.1.2 Managing Shadow Trees

Shadow trees, such as the one depicted in Figure 6-11, can grow to large proportions very quickly, and the kernel must prune them whenever possible. Pruning is done whenever the kernel realizes that an intermediate shadow object is no longer needed. Such an object is not needed in the following circumstances:

- If the system has copied all of a backing object's pages to the shadow object, the shadow no longer needs to reference the backing object, and instead can directly reference the next object in the chain.
- If a backing object is referenced only by a single shadow object, the two objects can be merged into one object.

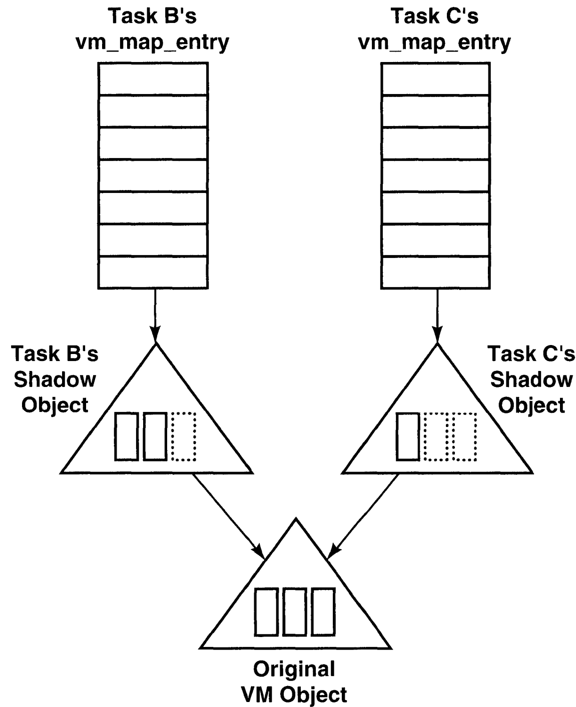
Consider again Figure 6-11. Note that task C's shadow object has already copied the sole page that is managed by its backing object. When task C accesses any other data, it must get that data from pages managed by the original VM object. In this instance, the system can simplify the shadow tree by having task C's shadow object reference the original VM object directly. Figure 6-12 shows the simplified shadow tree.

Figure 6–12. Pruning the Shadow Tree



Now that task C's shadow object references the original VM object directly, and its previous backing object (task B's former shadow object) has only one shadow object referencing it: task B's current shadow object. The system can further prune the tree by moving the backing object's first page to the shadow object and setting the shadow object to reference the original VM object directly. Figure 6-13 shows the result of this operation.

Figure 6–13. Pruning the Tree Further



The system attempts to prune the shadow tree each time it services a copy-on-write page fault. After the system allocates the new page for the shadow object and copies data to the page, the system checks the shadow object's backing object to see if pruning is possible. If the shadow object is the only object currently shadowing the backing object, the system merges the two objects together. If the shadow object no longer needs to reference the backing object because it has copied all of that object's pages, the system removes the shadow's reference to the backing object and sets the shadow to reference the next object in the chain.

6.3.2 Asymmetric Copy-on-Write

In symmetric copy-on-write, both tasks share unmodified data and store the modifications they have made in their respective shadow objects. The data managed by the original VM object is temporary, as is the data managed by the shadow objects. The system discards both the original and the modified data after the tasks have completed execution.

This section discusses assymetric copy-on-write, which is the kernel's mehchanism for managing the virtual copying of permanent data. Suppose that a task, task A, has mapped into its address space a VM object that manages permanent data, which continues to be stored on disk after the task has completed its execution. The task can modify this data and expects the system to write the changes back to the file. In this discussion, assume that task A is the only task that can modify the on-disk copy of the data.

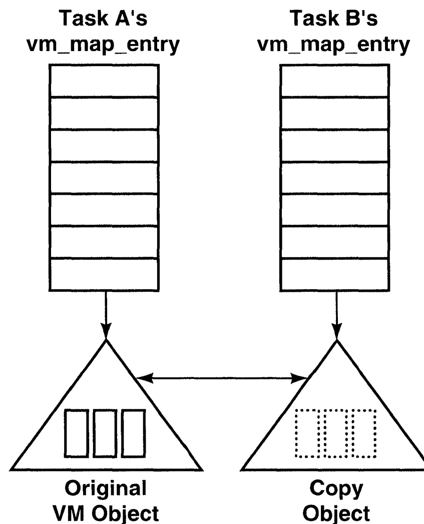
Task A's modifications to the file in secondary storage are made to an in-memory copy of the data, specifically, to data contained in resident pages. The system manages these pages via the VM object's list of resident pages. The system updates the file by sending the modified pages back to the VM object's associated paging object. Task A is able to modify the permanent data because it has that paging object's corresponding VM object mapped into its address space. In order for task A to retain its ability to modify this data, it must retain this mapping.

Suppose that task B, a task created by task A, has inherited a copy of the mapped file. Although the two tasks can, through a copy-on-write mechanism, share a virtual copy of the data, the system must continue to allow task A the exclusive right to modify the on-disk copy of the data. Task B may also modify data in the virtual copy, but its modifications are considered temporary. The kernel does not allow task B's changes to get written back to the permanent on-disk file.

As long as neither task modifies the data, they can share the virtual copy. However, suppose that task A wants to modify the data. To make its modifications permanent, task A must retain its mapping to the original VM object so that the system sends the modifications to the appropriate paging object. However, if task B has the original VM object mapped into its address space, it will see task A's modifications. If B is to avoid seeing A's changes, it must not map to the original VM object. OSF/1 uses the mechanism of the *copy object* to enable the tasks to share the data read-only.

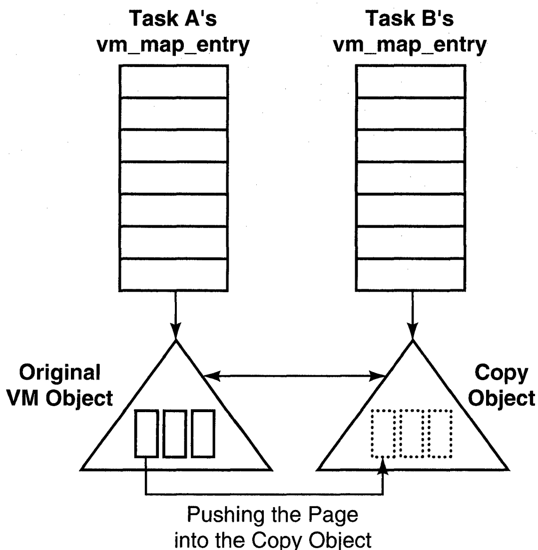
When the system initializes task B and sets up the virtual region that the task has inherited from task A, the system creates a VM copy object and maps this object to the region. As shown in Figure 6-14, this object is initially empty, but references the original object's pages. For example, if task B wants to read data from the virtual copy's first page, it reads the data from the original object using the copy object.

Figure 6–14. Tasks A and B Share Permanent Data Copy-on-Write



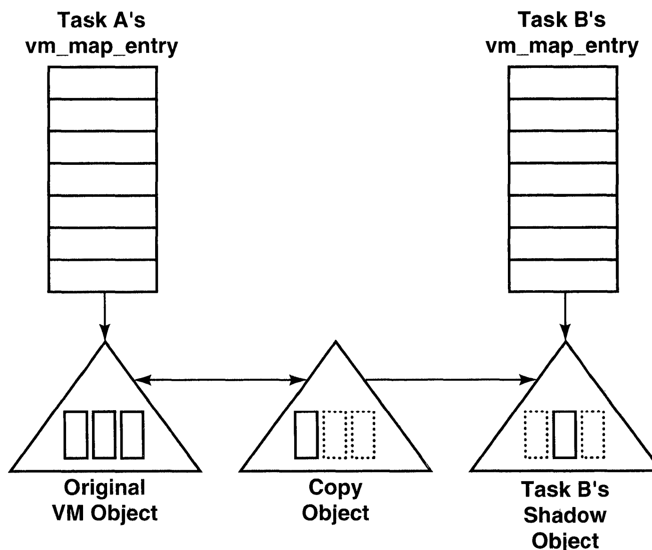
Suppose that task A wants to modify data contained in the original VM object's first page. Before the system allows this modification to take place, it allocates a new page, copies the data from the original page to the new page, and then inserts the new page into the copy object (Figure 6-15). Task B then references the new page, which contains unmodified data, and task A is free to modify the original page. Task B still reads any unmodified pages from the original VM object.

Figure 6–15. Task A Writes Data, Pushing a Page to the Copy Object



Suppose that task B wants to modify its virtual copy of the data, and the data it wants to modify is contained in the virtual copy's second page. As far as the system is concerned, task B has the data mapped copy-on-write. When task B attempts to write the data, the system initializes a shadow object, allocates a new page, copies the original data into the new page, and inserts the new page into the shadow object. The system then maps the shadow object to task B's address space. See Figure 6-16.

Figure 6–16. Task B Writes Data



As Figure 6-16 shows, the data modified by task B lies in the shadow object. If task A has modified the original data, it pushes an unmodified copy of the page to the copy object, and task B can read the data from there. If neither task modifies a page's data, task B reads the data from the original object using the shadow and copy objects.

This virtual copy mechanism is known as asymmetric copy-on-write because the system creates shadow objects only for child tasks. The parent task retains the right to have its modifications sent back to permanent storage, so it continues to map the original VM object after it has modified the data.

6.4 The Page Fault Handler and Copy-on-Write

When a thread faults on a page that is shared symmetrically, the kernel has to copy the page's data to a new page. In this case, the page fault handler has passed to **vm_fault_page( )** the shadow object that will hold the copied page. The actual page is contained in a backing object somewhere down the shadow chain. The routine searches the virtual-to-physical hash table and determines that the page is not resident. The routine then allocates a new resident page and begins searching for the backing page that contains the data. This search proceeds as follows:

1. The routine accesses the next VM object in the shadow chain using the current object's *shadow* field and determines the offset into this object. It then unlocks the current object and locks the next object. The newly allocated page is automatically marked busy so that another task faulting on the page will wait for it to fill instead of racing down the shadow chain.
2. The routine searches for the page in the new object by calling **vm_page_lookup( )** and passing to that routine the new VM object and offset.

This set of steps continues until the routine finds the page containing the data. When the routine finds the page, it copies its contents to the page allocated in the top object, then returns the page to the **vm_fault( )** routine. If the page is not found, the page fault handler initiates a pagein operation.

In the case of asymmetric copy-on-write, the task wants to write permanent data that is marked copy-on-write. The faulting task owns the data in the sense that its changes are eventually written back to the data's secondary storage entity. The other tasks that are sharing this data consider their copy of the data to be temporary, and any changes they make to the data are not written back to the secondary storage entity. Before the faulting task writes the data, it must "push" an unmodified copy of the data to the copy object. The **vm_fault_page( )** routine proceeds as follows:

1. It allocates a new resident page to hold the copied data. This page is associated with the copy object.
2. It copies the contents of the original page to the copy object's page.
3. Other tasks that are sharing the data may have referenced the original page and so have entered the page into their pmaps. These pmap

entries must be invalidated because the faulting task is about to write the data. The routine invalidates any other mapping by calling the **pmap_page_protect( )** routine.

6.5 Share Maps

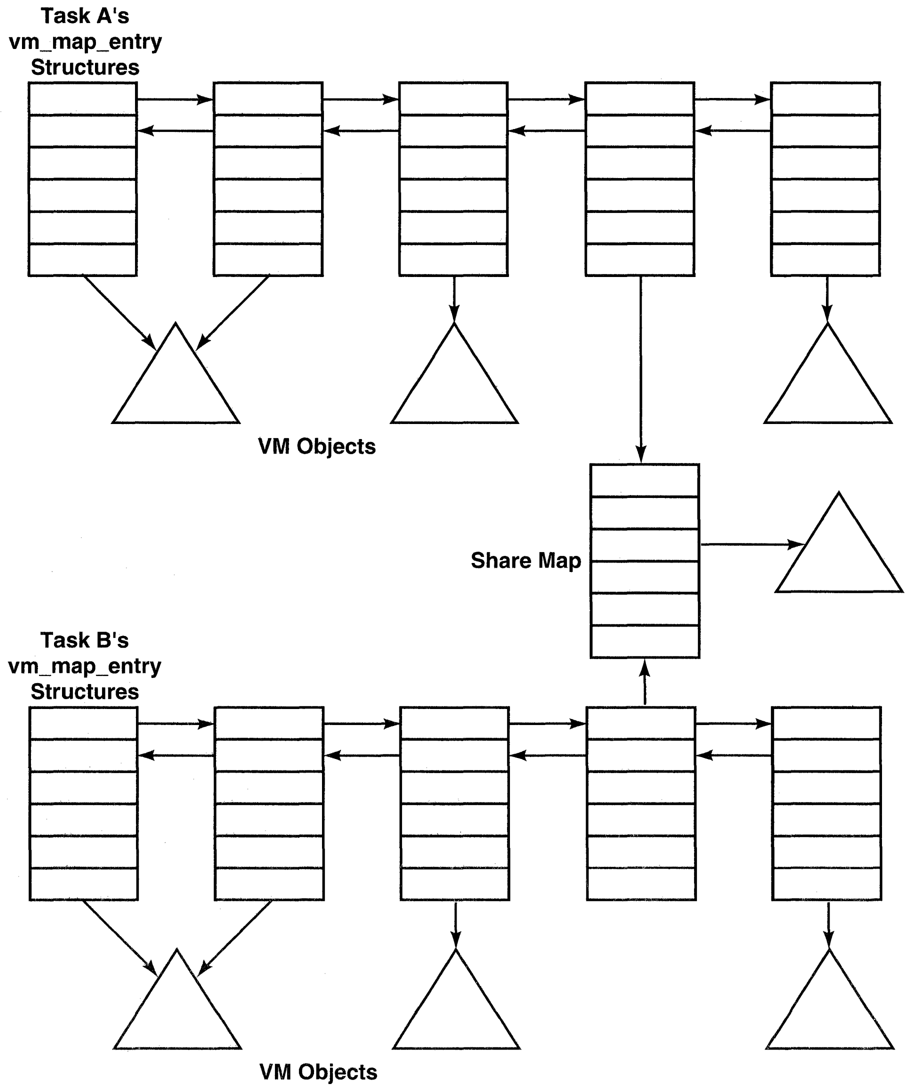
Two tasks can share access to a region of data. This is different from sharing data copy-on-write. When two tasks share access to data, each task sees changes made by the other.

A task that shares access to data with another task may want to share the data with a child task on a copy-on-write basis. Tasks cannot share access to data through a VM object because that structure cannot support copy-on-write and sharing at the same time. This sharing arrangement is implemented instead by a *share map* mechanism. A share map is an address map that is shared between two or more tasks.

Usually, true sharing can only take place between related tasks (an external pager can be implemented that allows unrelated tasks to share data, as discussed in Chapter 7). One task shares data with a child by allowing the child to inherit shared access to the data. The child task may then allow its children to inherit shared access, or it may allow its children to inherit a copy of the data.

Figure 6-17 outlines how a share map works.

Figure 6–17. A Share Map



Share maps are rare in OSF/1; they can be generated when a process maps a file with **mmap( )**, changes its protection with **vm_inherit( )**, and then forks another process.

6.6 Virtual Copy and Mach IPC

In Mach IPC, tasks can send messages of arbitrary size to one another. In traditional UNIX systems, transferring large amounts of data between processes is an expensive operation because it involves physically copying the data from one address space to another. In Mach, however, such operations are inexpensive because the data is passed copy-on-write.

OSF/1 implements *copy map objects* to support Mach IPC copy-on-write operations. A copy map object, which is not the same as the copy objects used to implement asymmetric copy-on-write, is an actual copy of a portion of a task's address map. Each copy map object contains an address map data structure, which is chained in double-linked fashion to address map entries that represent the copied address space.

The **vm_map_copyin( )** structure, which is used by the Mach IPC subsystem when a task sends out-of-line data to another task, duplicates a portion of a task's address space by creating a copy map object to represent the data being passed in the IPC message. The copy map object is subsequently inserted into the receiving task's address space. The copy map object may also be used to overwrite the task's address space.

6.7 The Kernel's Address Space

The OSF/1 kernel is implemented as a task, called the *kernel task*. Compared to most user process address spaces, the kernel task's address space is complicated. Among other things, it contains the kernel's executable text, the data structures it uses to represent system entities such as tasks, threads, and VM objects, kernel stacks for user-level threads, and so on. Like any other task, the kernel has an address map that describes its virtual address space. This map is called the *kernel map*.

Address maps, both kernel and user, are protected by locks. Address maps can be locked for reading, and separately for writing. Multiple threads can lock the address map for reading. A thread that has a map locked for writing has exclusive access to the map; other threads that may want to read the map cannot do so until the writing thread releases the lock.

On user process address maps, the lock mechanism does not significantly affect the performance of the application because the lock is not generally under contention. User address map locks are taken in two circumstances:

- The kernel is resolving a page fault with respect to the process and is examining the map's address map entries.
- The process is mapping or unmapping a VM object into or from the address space and is adding or removing an address map entry from the map.

In most user processes, these are not activities that generally overlap. When they do, the application's performance may be affected slightly.

However, if the kernel address map, with its varied regions of data, were locked by a single lock, many entirely unrelated threads would contend for it, and system performance would suffer. Therefore, the kernel map is divided into *submaps*, which can be locked separately from one another.

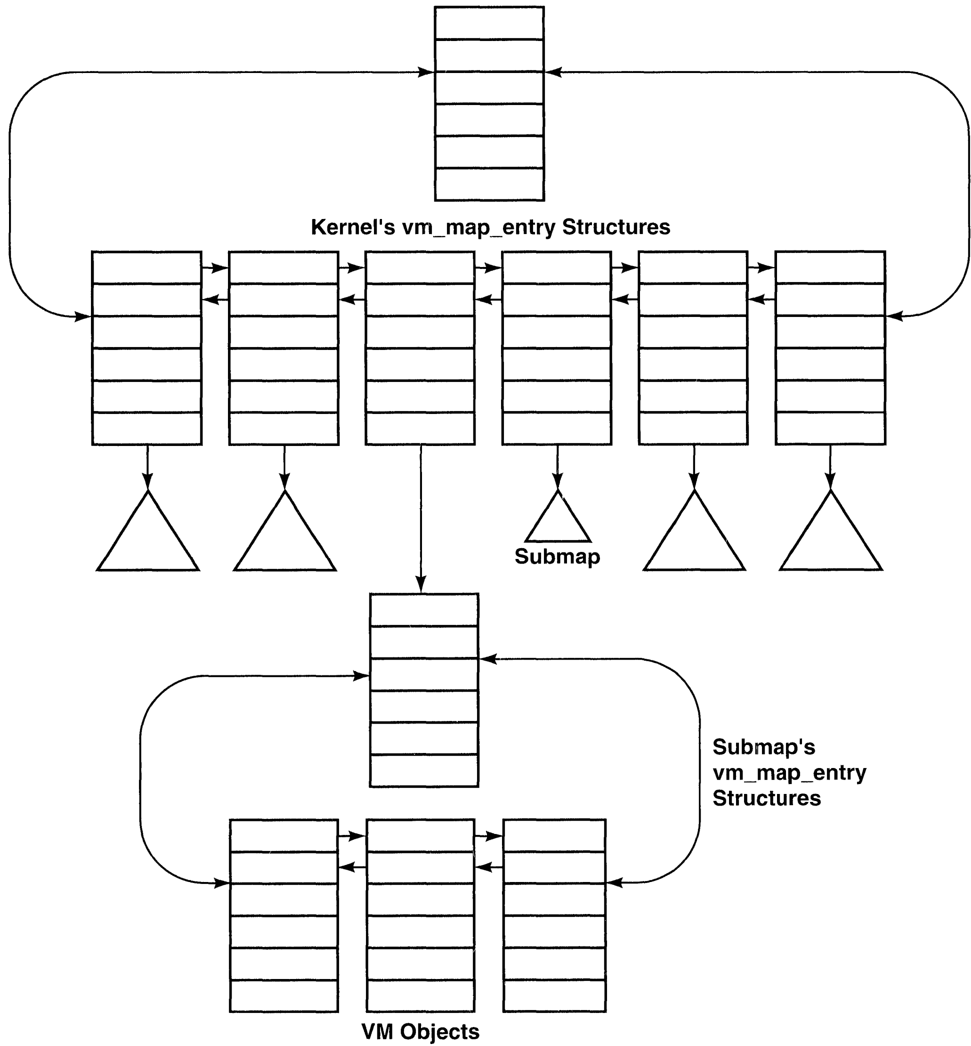
For example, the kernel's default pager, the vnode pager, executes in the context of the vnode pager submap. The vnode pager, with its multiple threads, often has occasion to lock its virtual address space. Because this address space is built into the kernel's address space, it would be inefficient for locks taken by the pager to prevent other threads from accessing other unrelated kernel data structures.

Some of the kernel's submaps manage wired-down memory. The default pager submap is in this category. Since the default pager must be able to immediately free memory resources, it cannot afford to generate page faults; its memory is always resident.

6.7.1 Submap Implementation

Submaps are implemented through address map data structures. The submap contains one or more address map entries that specify the address space it manages, and the submap is itself referenced by an address map entry in the parent map. See Figure 6-18.

Figure 6–18. The Kernel's Address Map with Submaps



6.8 Pmaps and the Pmap Module

A task's pmap is an abstraction of the machine-dependent data structures that the memory management unit (MMU) uses to perform address translation. Because all virtual-to-physical translations are maintained in the machine-independent data structures, a task's pmap serves as a cache of those translations.

In keeping with Mach's philosophy of lazy evaluation, the kernel manipulates pmaps only when it is absolutely necessary. Most VM operations are implemented through machine-independent code. When the machine-independent VM performs an operation that must be reflected in a task's pmap, it issues a call to the kernel's pmap module. This module, which contains all machine-dependent code associated with the VM system, implements the set of services required by the machine-independent VM.

The pmap module must implement the pmap data structures and the operations that manipulate these structures. The technique the pmap uses to manage its cache is specific to the memory management hardware. The data structures differ for single-level page tables (the DEC VAX), multilevel page tables (the Encore Multimax and the Motorola 68030), inverted page tables (the IBM RT/PC), paged segmented architectures (the Intel i386), translation caches (MIPS), and so on. Each of these machines has its own unique pmap data structures, but exports the same basic functional services.

6.8.1 The Pmap Functions

The routines and macros exported by the pmap module can be grouped according to their functions:

- Managing individual pmaps. This includes routines that create and destroy pmaps.
- Managing threads. This includes routines that install and update virtual-to-physical mappings.
- Performing global operations on multiple pmaps simultaneously.
- Manipulating physical memory. There are also optional routines that can be used by the pmap module to perform optimization operations.

6.8.1.1 Managing Individual Pmaps

The **pmap_create( )** routine is called when a task (and therefore a new address space) is created. It returns a handle for the pmap structure to the machine-independent code. This handle is used by all routines to specify which pmap to operate on.

The **pmap_kernel( )** call returns the handle for the pmap describing the kernel address space. It is used by routines in the virtual memory system to initialize and manage the kernel's address space.

The **pmap_reference( )** and **pmap_destroy( )** routines increment and decrement reference counts on pmaps. If decrementing the count puts it at 0 (zero), **pmap_destroy( )** deallocates the physical map and frees up its data structures.

6.8.1.2 Managing Threads

The pmap subsystem provides macros to help manage threads within processes. The **pmap_activate( )** routine invokes the **PMAP_ACTIVATE( )** macro whenever a CPU is dispatched to run a thread, and the new thread is in a different process (and therefore in a different pmap). The macro sets up any hardware context in the specified CPU's address translation hardware, ensuring that the mappings established in the specified pmap will be valid in the thread.

When **PMAP_ACTIVATE( )** completes, the specified pmap is active. The **PMAP_DEACTIVATE( )** macro does any cleanup required by the address-translation hardware, so that the mappings established in the specified pmap are no longer valid addresses.

The normal sequence during a thread context switch requiring a process context switch is to have **PMAP_DEACTIVATE( )** called, supplying the old process's pmap and the CPU; then **PMAP_ACTIVATE( )** is called, supplying the new process's pmap and the CPU; and then the process context is switched. For context switches between threads *within* a process, the **PMAP_CONTEXT( )** macro is called. This macro sets up any per-thread hardware state in the specified CPU's address-translation hardware, so that the mappings established in the specified pmap will be valid in the new thread.

6.8.1.3 Managing Address Ranges Within the Pmap

Changes in virtual address mappings are reflected back to the pmap module using address-space-specific calls. The **pmap_enter( )** service inserts a virtual-to-physical address mapping in the pmap (with the requested protection). It is the basic routine used to validate addresses. The **pmap_enter( )** routine is the only routine in the pmap that cannot be lazily evaluated. When **pmap_enter( )** returns, the specified mapping must exist in the map.

The **pmap_remove( )** routine removes a range of addresses from a pmap. It is the basic routine used to invalidate addresses. The **pmap_protect( )** routine limits the maximum allowed access for the specified range of virtual addresses in the specified pmap to the specified protection. If the specified protection is higher than the current protection for any currently valid page in the range, the protection for that page is not changed. Therefore, **pmap_protect( )** can never increase page protection.

Other routines that act on virtual addresses within a pmap are **pmap_extract( )**, **pmap_access( )**, and **pmap_change_wiring( )**. The **pmap_extract( )** call translates a virtual address within a specified pmap into the physical address to which that virtual address currently maps. Essentially, it simulates the operation of the address-translation hardware. The **pmap_access( )** call determines whether there is currently a valid mapping for the specified virtual address in the specified pmap. If a valid mapping exists, it is assumed that a reference to the address will not cause a page fault. The **pmap_change_wiring( )** routine changes the wiring of the physical page that contains the specified virtual address in the specified pmap. It is used to set the wiring attribute so that a reference to the page may or may not page fault.

6.8.1.4 Tracking Pages Mapped to Multiple Pmaps

There are kernel functions that affect multiple address spaces simultaneously, typically acting on all mappings of a physical page. The **pmap_page_protect( )** call limits the maximum allowed access for the specified page to the specified protection. This new protection applies to all

pmaps that have the page mapped. The **pmap_page_protect( )** routine has three important uses:

- It is used to remove write access from a page during copy-on-write operations.
- It is used to invalidate all accesses to a page when the page is being freed as a result of page replacement or object deletion.
- It is used to lock out all accesses of a specified page, in response to a request from an external memory manager that is managing that page. A memory manager might make this request when it is attempting to maintain cache coherence across a network.

In order to implement these global operations on multiple mappings, the pmap must maintain a list of physical-to-virtual mappings. Entries in this list (defined by the structure **pv_entry**) contain, for each active mapping of the page, the address of the pmap structure and the virtual address at which it is mapped. An array, whose address is held in **pv_head_table**, is allocated at initialization time to contain the first list entry for each physical page. If a page is mapped by no more than one process, there is no actual list, just an array element to indicate where it is mapped. As pages become shared by more than one process, the new list entries are allocated and linked.

Other routines that must walk this physical-to-virtual list to return information about a page include **pmap_is_modified( )** and **pmap_is_referenced( )**. These calls check the modified and referenced bits, respectively, for the specified physical page. The modified bit information is used by the pageout code to decide whether the contents of the page need to be written to backing storage before the page is freed. The referenced bit information is used by the page replacement algorithm to decide whether or not an inactive page is a candidate for being freed. Some address-translation hardware does not support reference bits. Since it may be too expensive to simulate a reference bit in software, these machines may simply always return FALSE from **pmap_is_referenced( )** and depend on **pmap_clear_reference( )** to remove all mappings for the specified page.

The **pmap_clear_modify( )** routine is used to reset the modified bits to a known value (for example, when the page has just been read in from backing storage). The **pmap_clear_reference( )** routine is used when the page is deactivated (that is, moved from the active list to the inactive list). If the page has been referenced, it is not freed but, rather, reactivated.

6.8.1.5 Services that Promote Optimization

The **pmap_copy_page()** and **pmap_zero_page()** calls actually manipulate physical memory. The **pmap_copy_page()** call is used whenever the kernel has to copy a page of data that may not have a virtual address in the kernel's address space. It copies a page of data from the source physical page frame to the destination physical page frame. This routine is used when a page must be copied because of a copy-on-write fault.

The **pmap_zero_page()** call clears the specified physical page frame by filling it with zeros. It is also used when the kernel has to clear a page that may not have a virtual address in the kernel's address space. Its most typical use is on the first user's reference to a page of storage allocated through **vm_allocate()**. These two routines can use the following optimizations:

- An implementation might permanently map the entire physical address space to a range of kernel virtual addresses.
- Some hardware types temporarily turn off address-translation during the copy or clearing operation.
- Some implementations dedicate some kernel address space for use in mapping physical pages to be copied or cleared.

Another call that promotes optimizations is **pmap_update()**, which synchronizes pmaps by telling the pmap module to perform any update operations that have been deferred. All calls that affect mapping, except **pmap_enter()**, may be delayed until **pmap_update()** is called. This is an example of lazy evaluation. **pmap_update()** is called only as needed to ensure that the state of the address-translation hardware is consistent with the virtual memory system data structures, so that a thread about to run will find a semantically correct address space.

A number of pmap calls are advisory, in that they supply information from machine-independent code to the pmap, which the pmap module can use as its optimization implementation dictates. The **pmap_copy()** call informs the pmap module that the specified range of virtual addresses in the destination map is to be mapped from the same physical pages with the same protections and wiring as the range specified in the source map. It is used to promote optimization in the **fork()** operation, where the child process's pmap is initialized to a copy of the parent's. The pmap module is not required to act on this information. The pages in the specified range will

eventually be copied as a result of the first fault on that page if the optimization is not used.

The **pmap_pageable( )** call informs the pmap module that the specified range of virtual addresses is to be wired (or else pageable). The hardware resources required to translate an address in this range (for example, its page table entries) must also be wired (or pageable). This call provides an efficient method for wiring the hardware resources for a group of pages simultaneously. Again, the pmap is not required to use this optimization advice. The pages will eventually be wired (or unwired) as a result of subsequent calls to **pmap_enter( )**.

The **pmap_collect( )** call is made by the thread swapout code when free memory is very tight. It frees as much memory as possible from the specified pmap. For example, memory holding page tables can be freed this way. The **pmap_collect( )** routine can invalidate mappings because the information necessary for reconstructing the pages is retained in the machine-independent code, allowing the page to be restored at page fault. The implementation must decide what to do with the optimization information.

6.8.2 The Shootdown of Translation Lookaside Buffers

A task's page tables reside in resident memory, but address translation would be prohibitively slow if the MMU had to reference resident memory for each translation operation. Most hardware architectures, therefore, optimize address translation by caching translations in the CPU's *translation lookaside buffer* (TLB). When a task references a virtual page whose page table entry is not cached in the TLB, the MMU loads the page table entry from resident memory into the TLB. As long as the entry remains in the TLB, the MMU can continue to translate references to the virtual page without having to access the page table entry in resident memory.

An entry that is cached in a TLB may be modified and then written back to the page table in resident memory. For example, if an entry is cached in the TLB and the thread on the CPU writes to the page represented by the entry, the hardware will update the modify bit in the cached entry. Subsequently, the entry will be written back to the page table in resident memory.

On architectures that provide TLBs, the operating system must make sure to keep the contents of the TLB synchronized with the contents of the page tables in memory. For example, suppose a particular page's page table entry has been cached in the TLB. If the kernel updates the page table entry in resident memory to change the page's protection from read/write to read-only, the cached page table entry becomes invalid and must be removed from the TLB so that the valid page table entry can be used.

The management of TLBs on shared memory multiprocessors is complicated by the fact that a thread on one processor may modify a page table entry that is loaded not only in its own TLB, but in the TLBs of other processors. The system must have a mechanism for maintaining the consistency of translations across the TLBs.

Ideally, the hardware would implement a mechanism that would allow processors to manipulate one another's TLBs. Unfortunately, most architectures do not provide such a mechanism. On these architectures, TLB consistency must be maintained by software.

The main problem that must be solved with respect to TLB consistency is one of timing. When a page table entry must be updated to reflect a change in the mapping, the update can be made before or after the TLBs are flushed. If the entry is updated after the TLBs are flushed, there is a chance that one or more of the TLBs may reload the entry before the entry has been updated. If the entry is updated before the TLBs are flushed, one of the processors may inadvertently overwrite the resident entry with the previous version of the entry.

These problems can be solved by a means of communication that allows a thread that is changing a pmap to stall all other processors that are using the pmap. While a processor is stalled, it cannot write TLB entries back to resident memory. When the initiating thread is sure that the other processors are stalled, it updates the pmap, and then unstalls the processors. When the processors become unstalled, they immediately flush their TLBs.

This algorithm is called a *TLB shootdown*. It is implemented within the pmap module, where it is divided into two portions:

- The code executed by the initiating thread, which sends interrupts to other CPUs if the current pmap operation might introduce an inconsistency in the pmap
- The code executed by responding threads, which receives interrupts and performs the operations required to keep the TLB consistent

The shootdown algorithm performs its operations by managing the following data structures:

- A list of processors that are currently performing address translation
- A list of processors that are idle
- For each pmap, a list of the processors that are using the pmap
- For each processor, an "action needed" flag that indicates the need for a TLB consistency operation, and buffers to hold pending consistency actions

A thread invokes the shootdown algorithm when it performs a pmap operation that could cause inconsistencies in the TLBs of other processors. The algorithm performs its operations in four phases:

a. The initiating thread places a consistency action request in the buffer of each processor currently using the pmap and sets the action needed flag of each processor. The initiator then sends an interrupt to each of the processors.

b. The responders receive the interrupts. Each responder removes itself from the list of active processors to wait for the initiating thread to update the pmap. This prevents the responders from attempting to read from or write to the pmap while the initiating thread is updating it.

c. After the responders have entered the waiting phase, the initiator updates the pmap.

d. When the initiator has finished updating the pmap, the responders invalidate their TLBs, clear their action needed flags, and place themselves on the list of active processors.

The previous description illustrates the basic structure of the algorithm; however, the actual algorithm is more complicated because it must account for situations inherent in multiprocessor environments.

Chapter 7

The Virtual Memory Subsystem: Memory Management

An operating system manages its memory resources through a memory management subsystem. Memory resources include the system's resident memory and secondary storage devices such as disk drives. It is the memory management subsystem's responsibility to allocate memory resources effectively among concurrently executing processes.

In the evolution of UNIX memory management, there have been two approaches to allocating resident memory resources: *swapping* and *demand paging*. In a swapping-based memory management subsystem, a process must be entirely resident to execute. The operating system moves entire processes between resident memory and secondary storage to achieve sharing.

In a demand paging system, a process can execute without having to be entirely loaded in resident memory. Those portions of a process that are not resident are kept in secondary storage and are *paged into* resident memory as needed. When the memory management system needs to reallocate page frames to other processes, it *pages out* process data to secondary storage to free the page frames.

Paging data in and out of resident memory requires more system overhead than swapping in and out entire processes because incremental paging operations require incremental disk accesses.

Demand paging systems are inherently susceptible to a condition known as *thrashing*. When resident memory resources drop to a certain level and the demand for resources is high—frequently the situation when many new processes are being created—the memory management system may not be able to free single pages quickly enough to satisfy the demands on memory. The system may spend a large amount of its time paging out pages at the expense of executing processes.

Thrashing can be alleviated somewhat with a hybrid demand paging/swapping memory management system. In systems of this type, demand paging memory management takes place when the system is under moderate load. Under heavy load, the memory management system can swap entire processes out to secondary storage. Most modern UNIX systems implement this hybrid memory management policy. OSF/1's memory management system is also a hybrid, although OSF/1's version of swapping differs from that of other UNIX systems.

7.1 Overview

Memory management in OSF/1 is based on the constructs of Mach memory management. In Mach, a task's virtual address space contains regions of allocated memory. Each of these regions is mapped to a memory object, and each memory object is managed by a memory manager. The memory manager implements paging operations on the object and performs these operations at the kernel's request.

The kernel requests paging operations by issuing IPC messages to the memory manager. For example, the page fault handler initiates the pagein of data by issuing a pagein request to the page's memory manager. The memory manager pages in the data by sending it to the kernel in a message.

Similarly, the kernel initiates the pageout of data by issuing a request to the data's memory manager. The memory manager handles the request by writing the data to secondary storage and freeing the page frame that contained the data. It is important to note that it is the memory manager's responsiblity to free the page; the kernel merely requests the pageout operation.

The Mach memory management model supports the implementation of memory managers that can implement paging operations on application-defined objects. Memory managers of this type execute in user space and are often referred to as *external memory managers*. They interact with the kernel through the *external memory manager interface*.

7.2 The Vnode Pager

Because external memory managers are not directly controlled by the kernel, the kernel cannot depend upon these managers responding to pageout requests on a timely basis. When an external memory manager fails to free pages as requested, the system's *default pager* becomes responsible for paging out and freeing the pages. The default pager is a *trusted pager* (it runs with superuser privileges) and is guaranteed to perform pageout operations promptly. The default memory manager is also responsible for performing paging operations on temporary memory objects.

The OSF/1 default pager is the vnode pager. It is called the vnode pager because it manages memory objects that are files or devices, which are represented by vnode data structures (see Chapter 3). In addition to functioning as the system's default pager, the vnode pager manages the pagein and pageout of permanent data (data that has been mapped into a process by the file mapping call **mmap()**). In its capacity as a manager of mapped file data, the vnode pager functions as an entity distinct from the system's default pager.

Although OSF/1 supports the implementation of other memory managers, the vnode pager is the only memory manager provided in OSF/1. Developers can use the kernel's external memory manager interface to implement other memory managers.

The vnode pager is a separate task whose address space is implemented as a submap of the kernel's address map. Since it is built into the kernel's address space, the vnode pager can directly access the vnode data structures that represent the files it implements paging operations on.

Another advantage to having the vnode pager reside in the kernel's address space involves the way pagein operations are initiated. When the page fault handler initiates the pagein of data that is managed by an external memory manager, it issues the request through an IPC message. The request cannot

be processed until the kernel switches context to the memory manager. However, because the vnode pager is embedded in the kernel's address space, the page fault handler can execute its pagein routines directly, thus saving the expense of the context switch.

The vnode pager is started during system initialization. At this time, the pager sets up its pageout-handling threads and assigns each thread a port set. Each thread's port set contains the ports that represent the memory objects managed by the thread. When the vnode pager initializes a new memory object, it places the object's port into one of the thread's port sets.

7.2.1 Paging Files

The OSF/1 system maintains one or more *paging files* that the vnode pager uses to back temporary data. Usually the paging files are set up during system initialization, but system administrators can add new paging files later with the **swapon** command.

A paging file may be either a raw disk partition or a file in a file system. The advantage of a raw disk paging file is that data can be written to it directly without having to proceed through the system's buffer cache. The advantage of the file system paging file is that its size can change dynamically; a size of a raw disk paging file is static.

In OSF/1, paging files can be assigned priorities. This feature allows a system to configure and prioritize its paging files based on the performance characteristics of its secondary storage devices. The vnode pager invokes the paging file selection algorithm when it needs to allocate space to back temporary memory. This algorithm cycles through all paging files with the highest priority in round robin order to spread allocations across the files. When there is no more space available in the files at the highest priority, the algorithm begins cycling through the files at the next priority.

In OSF/1, a paging file can have a priority ranging from 0 through 4, where 0 is the lowest priority and 4 the highest. A file's priority is set when it is initialized during the **swapon( )** system call.

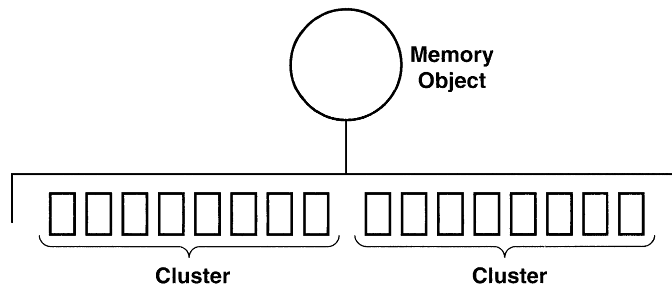
7.2.2 Page Clustering

The performance of the memory management system is critical to the performance of the entire operating system. Paging operations affect the system's performance because they require the system to read from or write to secondary storage. Such operations are inherently expensive; however, the expense of a given access operation does not depend on the amount of data transferred. In fact, the more data that is transferred in an operation, the more efficient the operation. It is more efficient to transfer two pages of data in an operation than it is to transfer one page.

To make paging operations more efficient, the vnode pager manages a memory object's pages in units called *page clusters*. As shown in Figure 7-1, a page cluster is a set of virtual pages that are adjacent to one another within their memory object and may also be stored contiguously in secondary memory. Pages that are stored contiguously can be read or written in a single operation.

For example, when the page fault handler needs to page in a particular page of data, it can read in the rest of the pages in the cluster with the same operation. Similarly, when the pageout daemon needs to write a particular page's contents to disk, it can simultaneously write other modified pages in the cluster.

Figure 7–1. Page Clusters



The number of pages in a cluster depends upon the type of data contained in the cluster. By default, data that is backed by paging files has a cluster size of four pages, although this can be changed by a system administrator. Currently, the default cluster size can be set to 1, 2, 4, or 8. In OSF/1, the

system's paging file cluster size is set when the first paging file is established; all subsequent paging files will have the same cluster size.

The cluster size for permanent data is established differently. Because the goal of cluster paging is to read or write as much contiguous data as possible with a single disk access, the cluster size for permanent data is a function of the file system block size. The block size is chosen because the file system usually allocates contiguous storage in blocks of this size.

The cluster size for a given region of permanent data is set when the data is mapped into the process's address space. At that time, the mapping operation divides the file system block size by the kernel's virtual page size to get the cluster size. For example, if the virtual page size is 4K and the file system's block size is 8K, each cluster will contain two virtual pages.

7.2.3 Allocating Clusters in Paging Files

In another example of lazy evaluation, the kernel allocates backing store for a given temporary data cluster only when that cluster is first paged out. This behavior enhances the system's performance because temporary data is often created and destroyed without having to be paged out.

When the pageout daemon initiates the pageout of temporary data that has never been paged out, the vnode pager chooses a paging file based on priority and availability and allocates space within the file to back the cluster. Because cluster allocation happens only when the cluster is being paged out for the first time, and a memory object's clusters are likely to be paged out at separate times, the clusters may end up being backed by different paging files.

Each paging file is represented by a **pager_file** data structure. The vnode pager uses this structure to manage the allocation of clusters in the file. The data structure includes the following:

- A pointer to the file's **vnode** data structure
- The number of allocated clusters currently contained in the file
- A map specifying the location of each allocated cluster within the file
- The number of free clusters currently contained in the file

- A hint that the vnode pager uses to begin searching for the next unallocated cluster

The vnode pager allocates a cluster in a paging file by examining its map of allocated clusters, beginning its search based on **pager_file**'s hint. When it finds an unallocated area in the paging file, it updates the paging file's map and search hint, and then begins paging out the data.

Although backing store for user-space anonymous memory is allocated only when it is required, the lazy evaluation strategy has dangerous implications when applied to the backing of thread kernel stacks. Consider what happens when the system attempts to page out a kernel stack cluster when there is no room left in the paging files. The operation fails, but the kernel does not discover the failure until it attempts to page in the cluster at a later time. The pagein attempt generates an unrecoverable page fault, which crashes the system.

Consequently, the kernel always allocates backing store for a thread's kernel stack when the thread is created. If the paging files are full at this time, the call to **thread_create( )** fails before initializing the thread. If the call to **thread_create( )** is successful, backing store for the thread's kernel stack is guaranteed to be in place.

7.2.4 Vnode Pager Memory Objects

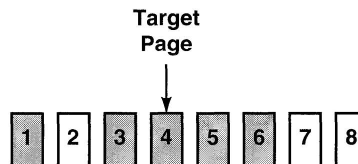
The vnode pager implements its memory objects via **vstruct** data structures. There are two types of **vstruct** structures: those that represent temporary memory objects, and those that represent mapped file memory objects. Each mapped file **vstruct** includes a pointer to the **vnode** data structure that represents the file. Since the clusters associated with a temporary memory object may be backed on different paging files, temporary memory object **vstruct** structures do not contain pointers to a single **vnode**. Instead, these **vstruct** structures contain *maps* that indicate which of the object's clusters have been paged out and where those clusters are located.

7.3 Cluster Paging Operations on Temporary Data

Temporary data is created in resident memory; it is placed in secondary storage only when it becomes subject to the page replacement algorithm. As mentioned previously, the vnode pager allocates secondary storage for a given temporary data cluster the first time one of the pages within the cluster needs to be reclaimed. At that time, the vnode pager allocates backing store for the entire cluster. Then the pager writes to that space the page being paged out, which is called the *target page*, and any other pages in the cluster that have been modified and are adjacent to the target page.

The page cluster shown in Figure 7-2 contains 8 pages. Pages 1, 3, 4, 5, 6, and 8 have been modified; pages 2 and 7 have not been modified. Suppose that none of the cluster's pages has been paged out, and that the pageout daemon has chosen the target page (page 4) for replacement. The vnode pager allocates backing store for the cluster, and then writes the target page and the adjacent modified pages; that is, pages 3, 4, 5, and 6. Pages 1 and 8 are not written back because they are not contiguous with the target page. When the write operation is complete, the target page is freed.

Figure 7–2. The Target Page



7.4 The Page Replacement Mechanism

The kernel manages the allocation of resident memory by maintaining three paging queues: the free page queue contains those pages that are available for allocation; the active queue contains pages that are allocated to processes; and the inactive page queue contains pages that are currently allocated but are candidates for being reclaimed.

The kernel maintains a free page threshold count. When the number of free pages drops below this threshold, the pageout daemon begins paging out

pages. This is driven by the page fault handler; when the page fault handler is invoked to page in data, it checks the number of free pages against the free page threshold and wakes up the pageout daemon if pages need to be replaced.

The pageout daemon processes inactive pages until it has produced enough free pages to meet the system's demands, or until it has depleted the inactive page queue. As it processes inactive pages, the pageout daemon moves pages from the active queue to the inactive queue to maintain a minimum number of pages in the inactive page queue. When it transfers a page from the active queue to the inactive queue, the daemon turns off the page's reference bit. If a process references the page before it reaches the head of the inactive queue, the reference bit will again be set, and the pageout daemon will transfer the page to the active queue instead of freeing it for reallocation. In this manner, the pageout daemon generally replaces pages on an approximately *least recently used* (LRU) basis.

The pageout daemon processes inactive pages a page at a time. If the page has not been referenced and has not been modified, the daemon can free it immediately by performing the following operations:

- Invoking the pmap module to remove all physical mappings to the page
- Removing the page's entry in the object/offset hash table
- Transferring the page to the tail of the free page queue

If the page has been modified, the pageout daemon cannot free the page. Instead, the daemon prepares the page for pageout.

7.4.1 Pageout of Data Managed by External Memory Managers

By default, all pageout operations in OSF/1 are managed by the vnode pager, which is capable of performing pageout operations on clusters. The original design of the Mach memory management system did not support cluster pageout; the pageout daemon assumed that all pageout operations involved single pages. OSF/1 retains the single-page mechanism because it is required to support the pageout of data to externally managed memory objects; that is, memory objects *not* managed by the vnode pager.

In all pageout operations, the pageout daemon initiates pageout by sending the data to its memory manager in an IPC message. The memory manager receives the data, writes the data to secondary storage, and frees the page. The operation proceeds as follows:

1. The pageout daemon creates a new temporary VM object that will be used to pass the page in the IPC message, transfers the page from its original object to the new object, and then invokes the IPC subsystem to pass the object to the memory manager.
2. The IPC subsystem maps the object into the memory manager's virtual address space. The memory manager copies the data from the page to the proper location in secondary storage.
3. The memory manager deallocates the message-passing object, thus freeing the page.

There are two important aspects to this operation. First, when the pageout daemon sends the page to the memory manager, it cannot guarantee that the memory manager will actually free the page. Note, however, that the page is passed to the memory manager in a temporary VM object; this object is managed by the vnode pager. If the memory manager does not free the page, the vnode pager will.

Second, while the page is being paged out, the pageout daemon must prevent the page from being paged in again until it is sure that the pageout operation is complete. This is accomplished by means of a *fictitious* page. Fictitious pages are **vm_page** data structures that do not point to actual page frames; they are used by the VM system to represent pages that are involved in paging operations.

The pageout daemon protects against premature pagein of the original page by

- Invoking the pmap module to remove ali physical mappings to the original page
- Allocating a fictitious page to represent the original page during the paging operation
- Removing the original page's entry in the resident page table with the fictitious page and marking the fictitious page as *busy*

Once the original page's physical mappings have been removed, any process that attempts to reference the page generates a page fault. When the page fault handler searches the resident page table for the page, it finds the

fictitious page is marked busy. The page fault handler must wait for the page to become unbusy; it goes to sleep to wait for that event.

When the pageout operation is complete, the pageout daemon removes the fictitious page from the resident page table and awakens all threads that were waiting for the page to become unbusy. These threads find that the page is no longer resident and initiate a pagein operation.

7.4.2 Pageout of Data Managed by the Vnode Pager

The pageout algorithm described in the previous section is not appropriate for the pageout of clusters. In that algorithm, the page being written is unavailable during the pageout operation. When paging out a cluster of pages, it is not appropriate to make an entire cluster of pages unavailable while the pageout operation proceeds because some of those pages may be in active use. The only page that should be unavailable during pageout is the page that will actually be freed; that is, the target page.

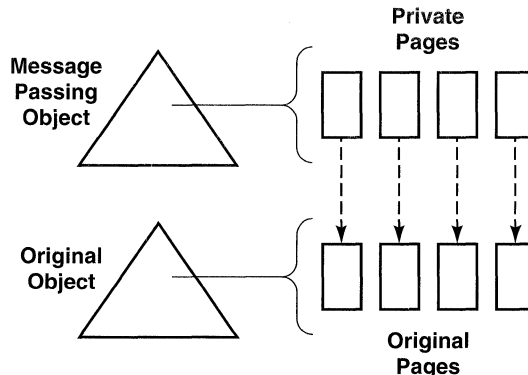
OSF implements cluster pageout using a technique called *cleaning-in-place*. In this technique, pages that are going to be written out are left in their original VM object so that they remain available during the pageout operation.

The cleaning-in-place mechanism works as follows:

1. The pageout daemon determines how much data will be written in the pageout request by finding all modified (dirty) pages adjacent to the target page.
2. The pageout daemon allocates a new VM object that it will use to pass the data to the vnode pager.
3. For each page that will be involved in the pageout operation, the daemon creates a *private page* and inserts the private page in the new VM object. Each private page points to the page it represents. Figure 7-3 shows this relationship.
4. The daemon makes the target page unavailable to other processes by removing its physical mappings and marking it as unavailable.
5. The daemon passes the new VM object to the vnode pager using IPC, and the object is mapped into the vnode pager's address space.

6. The vnode pager writes the data out to secondary storage and deallocates the object mapped in its address space when the operation completes.

Figure 7–3. Private Pages



7.5 The Page Fault Handler and Pagein of Clusters

Cluster pagein consists of the following interaction between the page fault handler and the vnode pager:

1. The page fault handler invokes the vnode pager's **vnode_pager_data_request_direct( )** routine, requesting the pagein of the target page. The fault handler may also page in other pages in the cluster. Normally, the fault handler will page in the target page and the next page in the cluster. A process may modify this behavior with the **madvise( )** system call.
2. The **vnode_pager_data_request_direct( )** routine allocates memory within the vnode pager's address space, and forces the allocation of physical pages to back this memory.
3. The routine reads the data in from secondary storage to the physical pages it has just allocated.
4. The page fault handler steals the pages from the vnode pager's address space and completes resolving the page fault. When it steals the pages, the page fault handler deallocates the VM object that was mapped into the vnode pager's address space.

When the page fault handler initiates a pagein operation, it must prevent other threads from trying to page in the same data. The page fault handler proceeds as follows: the handler allocates fictitious pages to represent the pages being paged in. (A fictitious page is a **vm_page** data structure that does not refer to an actual page frame.) The page fault handler places these pages in the resident page table to represent the pages being faulted in. The fictitious pages allow the handler to reserve locations in the resident page table without actually allocating physical page frames.

Then, if another thread attempts to page in the pages, it finds that the pages are "resident." However, the pages are not actually resident, they are being paged in. In this way, the page fault handler marks the pages busy so that other threads will realize that the pages are involved in a paging operation. When a thread sees that a page is busy, it puts itself to sleep to wait for the page to become unbusy.

When the vnode pager makes the data available, the page fault handler replaces the fictitious pages with the pages that the vnode pager allocated to receive the incoming data.

7.6 The Swapping Mechanism

With respect to traditional UNIX systems, the term *swapping* refers to the operation of copying the contents of a process's memory resources into secondary storage so that the resources can be reclaimed by the memory management system. For example, in 4.3BSD, the swapping mechanism forcibly pages out all resident pages associated with the process's data and user stack sections, and copies out the process's page tables, user structure, and kernel stack as well.

In OSF/1, the swapping mechanism reclaims the following resources from a process:

- The thread kernel stacks
- The process's resident pages
- The process's physical map

However, unlike the BSD swapper, which forcibly writes a process's resources to disk, the OSF/1 swapper relies on the pageout daemon to

actually free the resources. The OSF/1 swapper increases the number of pages available for swapout by moving a large number of pages from the kernel's active page queue to the inactive page queue.

7.6.1 Swapping Policy

The implementation of the swapping policy is cleanly separated from the implementation of the swapping mechanism to allow system vendors to easily provide their own swapping policies. The OSF/1 default swapping policy is very simple; it is expected that specific ports of OSF/1 will modify this policy to reflect the characteristics of the target environment.

Threads may be swapped voluntarily. The kernel swaps a thread voluntarily if it has been idle for at least 10 seconds. The pageout daemon initiates voluntary swapping when it begins processing the inactive page queue.

A process is swapped involuntarily when the kernel needs to quickly free memory. Involuntary swapping is performed by a kernel thread called the task swapper.

The kernel invokes the task swapper when paging demand warrants it. Under the default policy, the kernel checks the system's pageout rate once a second and calculates the average paging demand. The kernel monitors the amount of free memory using two thresholds: a *target* threshold, which specifies an amount of memory that the kernel would like to keep free (about 1.25%), and a *minimum* threshold, typically about 1% of memory. When resources fall below the minimum threshold, the kernel invokes the pageout daemon. The kernel initiates swapping when the amount of free memory remains both below the minimum threshold for 5 seconds and below the target threshold for 30 seconds.

The task swapper determines which process to swap out based on the number of pages resident and the amount of time the process has been resident. When a process is created, its task is queued on the list of swappable tasks. When the task swapper is invoked, it searches the list of swappable tasks. (Certain system tasks are not swappable; for example, the vnode pager.) By default, candidates for swapout have been resident for at least 6 seconds (this value is configurable). Of the tasks that have met this criterion, the task swapper will swap out the one that has the largest number of pages resident.

The swapper chooses a process to swap in based on the amount of time the process has been swapped out. By default, the kernel will swap in the process that has been out the longest, as long as the process has been swapped out for at least 6 seconds.

7.6.2 The Thread and Task Swappers

The OSF/1 swapping mechanism consists of a thread swapper and a task swapper. The thread swapper is responsible for recovering memory used by the thread's kernel stack. The task swapper is responsible for recovering memory being used by the task's pmap and its resident pages. The task swapper cannot swap a task until it has invoked the thread swapper to swap all of the task's threads.

7.6.2.1 Thread Swapping

The thread swapper is responsible for freeing a thread's kernel stack, the only pageable memory resource that is thread-specific. Normally, a thread's kernel stack is wired in memory so that it cannot be paged out. The thread swapper swaps a thread by unwiring its kernel stack so that the pages can be paged out, and then updating the thread's execution state to indicate that the thread is swapped out. The kernel stack pages are not forcibly paged out; they are merely made available for pageout.

The kernel swaps in a thread by rewiring its kernel stack. If any of the stack's pages have been paged out, they are paged in. Then the kernel wires the pages and updates the thread's state to indicate that it is swapped in.

There are two types of thread swapping: voluntary swapping and involuntary swapping. A thread may be swapped voluntarily if it has been idle for more than 10 seconds. This happens when the thread is waiting on an event. The kernel swaps in a voluntarily swapped thread when the event it is waiting for occurs.

The kernel involuntarily swaps threads as part of its task swapping operation. The task swapper suspends the task's threads, then invokes the

thread swapper to swap out the threads. The kernel swaps in involuntarily swapped threads when it swaps in the task.

7.6.2.2 Task Swapping

The task swapper performs the following operations when swapping out a task:

1. It swaps the task's threads.
2. It calls the pmap module to reclaim the memory being used by the task's pmap.
3. It determines which of the task's set of resident pages may be paged out and arranges for those pages to be paged out.

The implementation of the task-swapping mechanism is complicated by the fact that a task may share a number of its resident pages with other tasks. When the task swapper processes a task, it must be careful not to free pages that are actively being used by other tasks.

For each address map and each VM object, the kernel maintains a count of the tasks and threads that require the map or object to be resident. When this count drops to 0 (zero), the task swapper can swap the map's contents or the VM object's contents.

The task swapper begins trying to swap out a task's resident pages by decrementing the address map's residence count. If it decrements to 0 (zero) (as it usually does), the swapper goes through the map's entries and decrements the residence counts of its VM objects.

When a VM object's reference count goes to 0 (zero), the task swapper deactivates its pages by traversing the VM object's list of resident pages and placing those pages in the kernel's inactive page queue.

It is important to note that the task swapper does not swap out the task's data structure or the underlying VM object data structures.

When the kernel swaps in a task, none of the task's pages are explicitly brought in. They are faulted in as the task executes. Clustered paging improves the efficiency of restoring a swapped-in task's resources.

7.7 External Memory Managers

This section discusses issues related to the implementation of external memory managers. An external memory manager is a user-level program that implements paging operations on application-defined memory objects. For example, memory managers can be developed to provide services such as network shared memory and distributed databases that can be mapped into the address spaces of client programs that are executing on different machines.

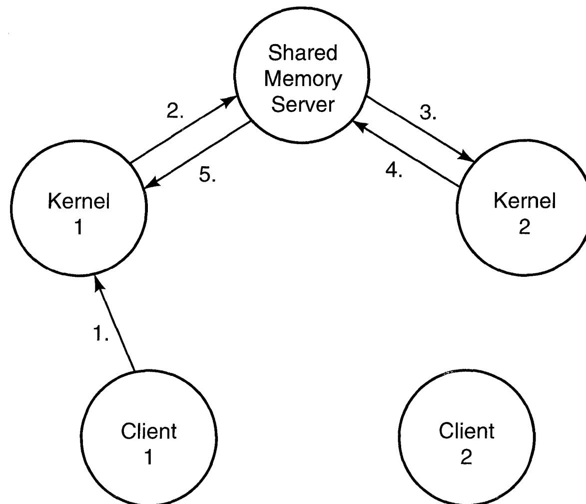
7.7.1 Example of an External Memory Manager: A Simple Shared Memory Server

The following example describes a simple shared memory server. This server allows tasks on separate machines to share read/write access to its memory objects, serializing write access to the data by allowing only one task at a time to write the data, and allowing multiple tasks to read the data when a write operation is not underway.

In Figure 7-4, two tasks running on separate machines have mapped the same memory object into their address spaces. When the clients are sharing read access to the memory object, the data is protected read-only on both machines. The figure shows how the shared memory server interacts with the machines' kernels when Client Task 1 attempts to write the data:

1. Client Task 1 generates a protection page fault because the data's page is protected read-only.
2. Kernel 1 sends a message to the shared memory server to request that Client Task 1 be granted write access to the page.
3. The shared memory server sends a message to Kernel 2 indicating that the page cached on that machine is about to become invalid.
4. Kernel 2 flushes the page and sends a message to the shared memory server when that operation is complete.
5. The shared memory server sends a message to Kernel 1 indicating that the protection of the page can now be changed, thus allowing Client Task 1 to write the data.

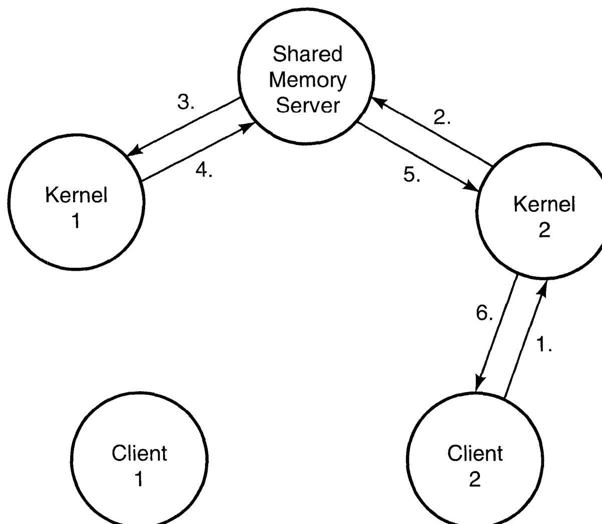
Figure 7–4. Shared Memory Server Write Operation



Suppose that Client Task 2 subsequently attempts to read the page that is being written. As shown in Figure 7-5, Client Task 2 initiates the following operations:

1. Client Task 2 generates a page fault because the page was flushed when Client Task 1 attempted to write it.
2. Kernel 2 sends a message to the shared memory server requesting a pagein operation.
3. The shared memory server sends a message to Kernel 1 indicating that the kernel should pageout the page and change the page's protection to read-only.
4. Kernel 1 changes the page's protection and allows the shared memory server to pageout the page.
5. The shared memory server now has a valid version of the page; the server provides the page to Kernel 2 to resolve Client 2's page fault.
6. Client 2 reads the data.

Figure 7–5. Shared Memory Server Read Operation



7.7.2 The External Memory Management Interface

The interactions that occur between the kernel and an external memory manager are implemented through the external memory manager interface. The routines specified by this interface can be categorized as follows:

- The *mapping* routine. Applications use the mapping routine **vm_map( )** to map memory objects into their address spaces. An application must acquire access to a memory object before mapping the object into its address space.

- The *memory object management* interface. An external memory manager must provide a variety of calls the kernel can use to access memory objects. These calls include the following:

  memory_object_init( )

  Notifies the memory manager that the memory object has been mapped into a task's address space. This routine establishes communication between the kernel and the memory manager; the kernel allocates a control port for the memory object that the memory manager can use to send management requests to the kernel.

memory_object_terminate( )
The kernel calls this routine when the memory object has been deallocated from the task's address space.

memory_object_data_request( )
Reads the contents of a memory object. The kernel calls this routine to resolve page faults on the memory object.

memory_object_data_write( )
Writes modified pages back to the memory object. The kernel calls this routine to page out the memory object's contents.

memory_object_data_unlock( )
Requests that the memory manager give the kernel permission to change the protection of the memory object's contents. For example, the kernel would call this routine if a task wanted to write data that was protected read-only. Not all memory managers need to implement this routine.

- The *cache management* interface. The kernel implements the routines in this interface to allow memory managers to maintain the contents of memory objects that are cached in resident memory. These routines include the following:

memory_object_data_provided( )
Supplies the requested page of data. This routine compliments the **memory_object_data_request( )** routine; the memory manager uses this routine to provide data in response to a page fault.

memory_object_data_error( )
Indicates to the kernel that the kernel's request for access to data cannot be granted. The data may not exist, or the kernel may be attempting to access it in a way that violates its current level of protection.

memory_object_lock_request( )
Requests that the kernel change the protection value of the pages cached in resident memory. A memory manager can use this routine to maintain consistency of data on multiple hosts.

The operations that constitute the external memory management interface are implemented with the IPC subsystem, but are not implemented as synchronous remote procedure calls. The operations occur asynchronously. For example, the kernel requests the pagein and pageout of data by sending a message to the data's memory manager, but the kernel does not wait for the memory manager to respond.

Chapter 8

The OSF/1 Program Loader

In OSF/1, each program executes in a virtual address space. This address space contains the text of all the subroutines and library routines the program requires. Before a given program can execute, the system must create an address space for it and map the program's text and data into the address space. Setting up the address space and mapping in the text and data is the responsibility of the system's program loader.

In traditional UNIX systems, programs are loaded with the **exec()** system call. **exec()** can load a program for execution only if the program's executable image is *absolute*; that is, has no *external references* and is *bound* to an address space (these terms are discussed in the following section).

Unlike traditional UNIX systems, OSF/1 allows programs to be loaded with unresolved references. This feature of the OSF/1 loader allows the system to support *shared libraries*.

When **exec()** loads a program in OSF/1, the kernel determines whether or not the program contains unresolved references. If it does not, **exec()** loads the program into a process's virtual address space and turns control over to the program's entry point. If the program does contain unresolved references, **exec()** loads the program loader into the process's address space, passes the program's name to the loader, and turns control over to the loader's entry point. The loader, which runs in user mode, finishes resolving

the program's references, loads the program into the process address space, and turns control over to the program's entry point.

In addition to supporting shared libraries, OSF/1's loader provides other features not available in traditional UNIX environments:

- The loader handles multiple object file formats.
- The loader provides an application interface that allows programs to explicitly load and unload modules to and from their address spaces.
- The kernel can use the loader to dynamically load modules into its address space while the kernel is active. This feature allows system managers to add components such as new device drivers and network protocols to the kernel without having to shut down, reconfigure, and reboot the system.

This chapter begins with a discussion of concepts associated with program loading. This discussion covers the following topics:

- Object modules and libraries
- External reference resolution
- Relocation
- Shared libraries

If you are familiar with these concepts, you may want to skip the following section and continue with the rest of the chapter.

8.1 Conceptual Background

Translating a source file into machine code involves two steps. First, the compiler translates the source code to assembler code and deposits the code in an assembler code file. When the compiler completes its translation, the assembler translates the assembler code file into object code and deposits that code into an object file.

Although object code is code that the machine can execute, an object file is not in itself executable if it contains *external references*; that is, references to variables and subroutines that are defined in other source files. For example, references to routines in the Standard C Library (**printf()**, for example) are external references. Before a program can be executed, the

code that defines its external references must be found and merged with the program. The merging of a program's object files into one executable file is referred to as *linking* or *binding*.

8.1.1 Linking

In the UNIX programming environment, program linking is traditionally performed by the **ld** utility. This utility can be used separately to link existing object files, but it is also called implicitly by the compiler to link the program being compiled.

The linker links a program using three operations: it resolves external references, it relocates the program's code, and it patches subroutine and global variable references. The linker resolves a program's external references by determining which library modules are required and merging those modules with the rest of the program's object code to form a single object file.

The linker relocates the program by binding the code to an address space so that the definition of each subroutine and global variable has a fixed address in the program's address space.

Once the relocation operation is complete and all of the program's subroutine and global variable definitions have fixed addresses, the linker patches each of the program's references to a subroutine or global variable by replacing the reference with the address of the corresponding definition.

A program that has been linked so that it no longer contains any external references is called an *absolute load module*. An absolute load module can be loaded into a process's virtual address space at a specific address and executed without additional processing.

In traditional UNIX, only *absolute execution images* can be loaded for execution. This means that each program must have its own copy of the library modules that are required to create the absolute executable image. For example, any program that uses the Standard C Library routines **gets( )**, **printf( )**, and **strlen( )** must contain its own copy of the C library modules required to implement these library routines.

8.1.2 Shared Libraries

In OSF/1, libraries can be simultaneously mapped into multiple address spaces; any process that requires one or more routines from a library can map the library into its address space when the process is loaded into memory. Libraries that can be mapped into multiple address spaces are called *shared libraries*.

In order to support shared libraries, a system must be able to complete linking a given program at load time, when the program is loaded into a process's virtual address space to begin execution. Traditional UNIX systems cannot support shared libraries because they cannot link programs at load time. OSF/1, on the other hand, supports shared libraries because its program loader can complete linking a program at load time.

8.1.3 The OSF/1 ld Command

In traditonal UNIX, the **ld** command, in addition to combining a program's object files, extracts those modules from the Standard C Library that are required to resolve the program's external references and links them into the program, creating the program's absolute executable image.

In OSF/1, the **ld** command does not produce an absolute load module if the program references routines from a shared library. Instead, **ld** produces a module that can be further processed at load time. When the loader loads such a module for execution, it finishes linking the program, and then maps the program into the process's address space.

8.1.4 Object Files and Object File Formats

As mentioned earlier, object files are typically produced when the assembler translates assembler code modules into object code. The typical object file has several sections. Some of these sections contain text and data and are referred to as "regions." The other sections contain information about the object file that the linker uses when linking the object file to other modules

and that the loader uses to map the object file's regions into the address space. This information includes the following:

- A list of the symbols the object file's code exports. This information is referred to as the file's *exported symbols*.
- A list of the symbols the object file's code needs to import. This information is referred to as the file's *imported symbols*.
- A *relocation dictionary* that specifies locations within the regions that need to be patched after relocation.
- For each region, information about the region's size and protection attributes. The loader uses this information when mapping the region into the address space.

The organization of an object file's contents is referred to as the file's *object file format*. There are many different object file formats. To successfully link and load a set of object files, the linker and loader must understand how the files are formatted so that they can extract the information they need to perform their operations.

Traditionally, UNIX systems were restricted to linking and loading a single format type because the format-dependent aspects of linking and loading were built directly into the **ld** utility and the **exec( )** system call.

In OSF/1, all format-dependent operations have been abstracted to facilitate the addition of new object file formats. OSF/1 supports the loading of multiple object file formats and can load the following object file types:

a.out The object file format supported by 4.3BSD

COFF The Common Object File Format used in the System V environment

OSF/ROSE The object file format developed at OSF that provides support for shared libraries in OSF/1

8.2 Overview of the Program Loading Architecture in OSF/1

The architecture of program loading in OSF/1 differs significantly from that of traditional UNIX systems for two reasons: the architecture supports the loading of programs with unresolved references, and the architecture can handle multiple object file formats.

8.2.1 The Architecture of exec( ) in OSF/1

When called in OSF/1, the **exec( )** system call must determine the module's object file format and must determine whether or not the module is absolute.

When the **exec( )** call loads an absolute module, the operations it performs to map the module's text and data into the address space and initialize the program's hardware state are format-dependent operations. In traditional UNIX systems, the code that performs these format-dependent operations is embedded within the **exec( )** code, and the **exec( )** code handles only one object file format. In order to support loading multiple file formats, **exec( )** must be able to recognize a program's object format and select the code it will use to perform the operations that are specific to the format type. In OSF/1, **exec( )** makes this selection with the **exec** switch.

8.2.1.1 The exec Switch

The **exec** switch is a globally available kernel data structure that implements a table of format-dependent routine vectors. Each entry within the table specifies a set of routines that are associated with a particular object file format. Each routine set (or vector) is referred to as the format-type's *file format manager*.

Each format manager consists of the following routine types:

recognizer Recognizes modules that have the manager's format type. The **exec( )** call attempts to recognize a program's format by cycling through the **exec** switch, trying each manager's

recognizer routine until it finds the manager that recongizes the program's format.

getloader If the *recognizer* routine indicates that the program needs to be processed by the loader, reformats (if necessary) the arguments supplied with the call to **exec( )**, and then prepares the user space loader for execution.

getxfile Maps the program into the process's address space.

setregs Initializes the hardware state to allow the program's execution to begin. The *setregs* routine is machine-specific as well as format-specific.

ungetxfile Removes the program's mapping from the current address space. The loader calls the *ungetxfile* routine to deallocate the current task's address space before calling the *getxfile* routine. All regions except those marked as **keep_on_exec** are deallocated.

8.2.1.2 The exec( ) Algorithm

The following list describes the algorithm used by **exec( )** to perform its operations:

1. Read the object file's header into a buffer.
2. Through the **exec** switch, cycle through all known format manager recognizer routines until the file's format is recognized.
3. If the object file's header indicates that the file is absolute:
    a. Unmap all regions in the address space.
    b. Map the object file's regions into the address space.
    c. Set the register state to execute the program.
4. If the object file's header specifies that the file needs to be processed by the user space loader:
    a. Read the loader's object file header into a buffer.
    b. Through the **exec** switch, cycle through all known format manager recognizer routines until the loader's format is recognized.

c. Unmap all regions in the address space.

d. Map the loader into the address space.

e. Set the register state to execute the loader.

8.2.2 The Loader's Architecture

The loader is a separate object module that resides in the user process's address space. It is loaded into the address space at a fixed location when the **exec( )** system call determines that the program requires the attention of the loader.

After being loaded and receiving control from the kernel, the loader loads the program and initializes it for execution by performing the operations outlined as follows:

1. Resolve the program's imported symbols by generating a list of the modules that will need to be mapped into the address space.
2. Use format manager routines to map the modules into the address space.
3. Use format manager routines to relocate those modules that need to be relocated.
4. Call initialization routines for any modules that specify them.
5. Jump to the program's entry point.

The mapping, relocation, and module initialization operations performed by the loader are format-dependent, and like **exec( )**, the user space loader maintains a table of routine vectors, each of which specifies mapping, relocation, and initialization routines. This table, which is referred to as the *loader switch*, resides within the loader's context and so resides in the process's user space with the loader. See Section 8.5 for more discussion about the loader switch.

8.3 The Symbol Resolution Policy

Because the loader's primary function is to resolve a program's symbols at load time, it must have a symbol resolution policy for matching each unresolved imported symbol in a module to a symbol exported by one of the known modules. In OSF/1, loader symbol management is based on the notion of *packages*.

A package is an object that exports symbols, and as such, packages can be thought of as abstractions of libraries. Each imported symbol that is not resolved at link time is represented by a *<package name, symbol name>* pair. The package name specifies the object that exports the symbol. Usually, a package represents a full library, but the package facility allows developers to divide libraries into multiple packages.

The package facility was designed with the following goals in mind:

- Imported symbols should not be bound to library pathnames. A program should not depend on the location of libraries in the file system (which may vary from system to system) in order to load correctly.
- Symbol name conflicts should be avoided without causing unnecessary restrictions on the use of symbol names by libraries. Each imported symbol in a program must resolve unambiguously to a symbol exported by a library at load time. Because imported symbols are not bound to library pathnames, conflicts are possible if more than one library in the loader's resolution path exports the same symbol name. Such conflicts should be avoided without requiring that exported symbol names be unique.
- Symbol resolution should be flexible but robust. It should be possible to control the symbol resolution path at compile, link, installation, and load time. The programmer and installer of a program should determine the default resolution path. Users should be able to run programs without worrying about symbol resolution, but users should also be able to alter symbol resolutions at load time when necessary. For example, when debugging a new version of a library routine, the programmer should be able to force programs to use the new entry point rather than the old one.

8.3.1 Using Packages

Generally, package names are attached to symbols at link time. In the following example, the **ld** command is used to create a shared library containing two routines: **subr1()** and **subr2()**. The source code for these routines resides in the **subrs.c** file. The following command generates the shared library:

```
% ld -R -o subrs.so subrs.o -export subrs_package:subr1,subr2
```

The **-export** flag causes the linker to create the **subrs_package** package, which contains the symbols **subr1** and **subr2**. The name of the package and the identity of its routines is recorded in the exported symbol table of the output file **subrs.so**. (In OSF/1, all shared library files end with **.so** by convention.)

Suppose that the file **test_subrs.c** contains code that tests the new shared library. When the **test_subrs.o** module is linked to **subrs.so** as follows, the linker searches for the package or packages within **subrs.so** that resolve external references:

```
% ld -o test_subrs test_subrs.o subrs.so -lc
```

Instead of loading the resolving modules into the output file, the linker replaces each imported symbol with the tuple *<symbol,package_name>*. For example, suppose that the imported symbol table in the unlinked **test_subrs.o** file contains the symbols **subr1** and **subr2**; after the linking operation, the imported symbol table for object file **test_subrs** contains the symbols **<subr1,subrs_package>** and **<subr2,subr_package>**.

When the loader links a given module to libraries at load time, the loader derives the package name for each imported symbol from the package name attached to the corresponding exported symbol in the library.

The two-dimensional symbol namespace provided by packages avoids symbol name conflicts when more than one library exports the same symbol. All that is required is that each symbol be unique within a package and that package names be unique across the system. Because each imported symbol includes a package, each imported symbol can then be resolved unambiguously to the correct exported symbol.

8.3.2 Package Tables

When the loader resolves imported symbols, it needs to find the library module that contains each package. This information is kept in a set of *package tables*. A package table is a set of mappings, each of which maps a package to a library pathname.

There are various types of shared libraries, and correspondingly there are various types of packages. For example, OSF/1 provides a shared library version of the standard C libraries. The system makes these libraries available to all programs with the system's *global installed packages table* (global IPT).

As described in Section 8.8, a system administrator can use the **lib_admin** command to install global libraries. The command completes the installation operation by writing the system's global installed packages table to disk. When the loader bootstraps itself into a process's address space, it maps the global installed packages table into the process's address space.

A program may require access to libraries that are not globally available, or may want to override a symbol's mapping in the global table with a mapping to an alternative version of the global routine. For example, a developer might want to test a new version of **printf( )** and still have access to the other C library routines that are globally available. In such instances, the loader builds a *private installed packages table* (private IPT) that maps symbol names to private libraries. This table is created and managed with the built-in shell command **inlib**. This command creates the table in the current process's address space in a region of memory that is not overwritten during calls to **exec( )**. Consequently, a process inherits its private installed packages table from its parent process.

Private packages are installed and used as follows:

1. The developer uses the **inlib** command to install one or more packages into the shell's private packages table.

2. The developer starts the program from the shell.

3. The program inherits the private packages table from the shell and the loader uses it during symbol resolution.

A program may itself export packages, or it may dynamically load packages using the loader's application interface. Under these circumstances, the loader creates within its context a *loaded packages table* (LPT) that maps symbol names to packages that have already been loaded. This table can be used to resolve symbols used by modules that may be loaded by subsequent calls to the loader's **load()** interface (discussed in Section 8.11).

During the loader's symbol resolution phase, it searches the packages tables in the following order: loaded package table, private package table, global package table.

8.4 The Loader Context

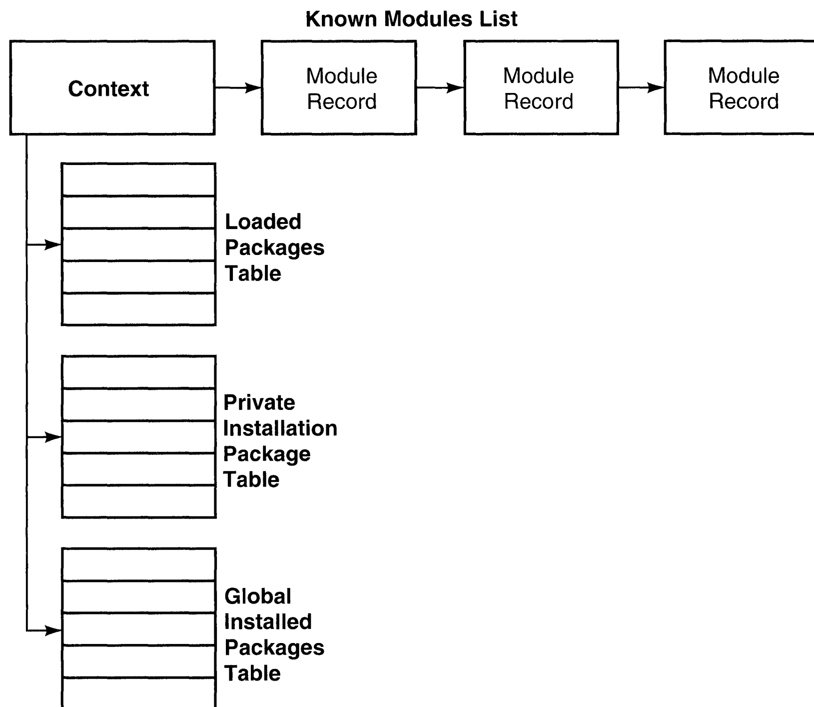
To construct an executable image of a program, the loader must find and load all the modules and carry out the symbol manipulation required to link all the imported symbols with exported symbols. This requires allocating memory for each region of each module, constructing a memory image of each region, and building tables of imported and exported symbols. Constructing a region's memory image requires writing or mapping in text, initializing data values, and filling in the correct addresses for all references. Filling in the correct addresses may require both looking up symbol values in the symbol tables, and, for relocatable code, adjusting relocatable addresses to reflect the actual location of code in memory.

The format-independent loader manages these operations by maintaining a set of per-process, per-module, and per-region data structures that are collectively referred to as the loader's *context*.

The loader context is a dynamic entity. The context is initialized when the kernel loads the loader into the process's address space. For example, the package tables of the context are set up when the loader context is initialized. The context changes as the loader performs the symbol resolution, mapping, and relocation operations.

Figure 8-1 shows a simplified version of the data structures that make up a loader context.

Figure 8–1. The Loader Context



Most processes have only one loader context, containing information about the modules loaded in that process. A special process, such as the kernel load server, however, maintains an additional loader context containing information about modules loaded into the kernel. There is also a loader context for the preloaded libraries and dynamically loaded format-dependent managers.

8.4.1 Module Records

During the symbol resolution phase, the loader creates a record for each module that it will map into the process's address space. The records that represent modules that are being loaded in the process's address space are placed in the context's *known modules list*. The order in which modules are placed on this list is important because it determines the order of symbol resolution. This is discussed further in Section 8.4.2.

When the loader has identified all of the modules that will be mapped into the process's address space, it traverses the known modules list three times, first to map each module's region into the address space, then to relocate each module's code and data, and finally to perform any module-specific initialization routines.

Each module record includes the following information:

- A pointer to the format-dependent routines that the loader will use to map, relocate, and initialize this module. This pointer is set when the loader creates the region record. At this time, the loader cycles through the loader switch entries and executes each recognizer function until it matches a format-dependent manager with the module's object format. The loader then initializes the module record's pointer to point to the manager's routines in the loader switch.
- A list of *region records*, each of which represents a region within the module that will be mapped into the address space during the mapping phase. The loader fills in these records using the format manager's *map_region* routine during the mapping phase, and the records are used by the format manager's *relocation* routine during the relocation phase.
- A list of the packages the module depends on. This is the module's *imported packages* list. The loader uses this list during the symbol resolution phase.
- A list of the module's imported symbols.
- An *exported packages* list if the module exports one or more packages.

8.4.2 Building the Known Modules List

As mentioned previously, the order in which module records are placed on this list affects the order of symbol resolution. When the loader begins loading a program into an empty address space, it creates a module record for the program's object module. At this point, this is the only record on the known modules list. The loader then begins building the rest of the list.

The loader constructs the known modules list iteratively, resolving the current module's imported symbols by creating module records for the modules that export the symbols and putting the module records on the list. The loader then moves to the next module record on the list and resolves that module's imported symbols, adding additional module records to the list as required. The procedure is repeated until the loader has resolved the symbols of all the modules that have records in the known modules list.

Figures 8-2, 8-3, and 8-4 illustrate this procedure. In Figure 8-2, the known modules list contains the module record for the loader (always the first module record on the list), and a module record for **prog.o**, an object file that contains two unresolved references: **subr1,packageA**, and **subr7,packageB**. In the example, **packageA** is exported by **sharelibA.so**, and **packageB** is exported by **sharelibB.so**. Figure 8-3 shows the known modules list after the loader has resolved **prog.o**'s imported symbols.

Figure 8–2. Known Modules List Example 1

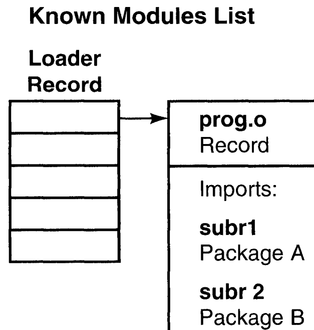
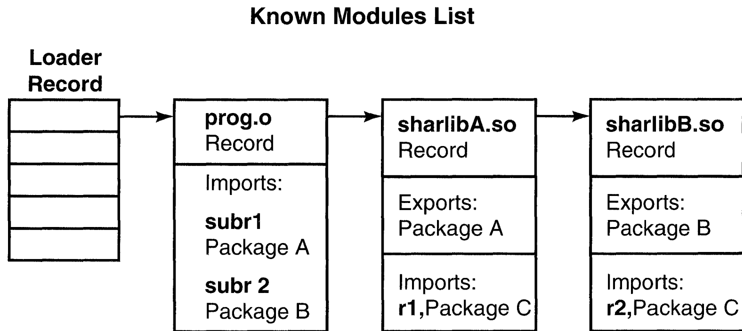
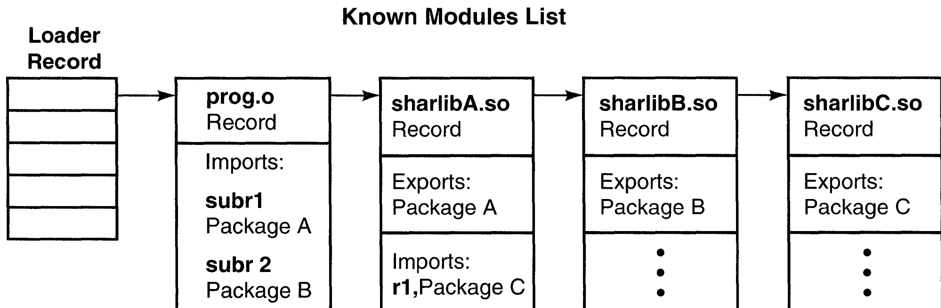


Figure 8–3. Known Modules List Example 2



When the loader finishes resolving the imported symbols for **prog.o**, it moves to the next module record on the known modules list and resolves any imported symbols specified by that module. As shown in Figure 8-3, the module record for **sharelibA.so** contains the imported symbol **r1,packageC**, which is exported by the module **sharelibC.so**. Figure 8-4 shows the state of the known modules list when the loader has finished resolving **sharelibA.so**'s imported symbol.

Figure 8–4. Known Modules List Example 3



8.5 The Loader Switch and Format-Dependent Managers

The loader switch is the primary interface between the format-independent and format-dependent portions of the loader. The switch is a set of data structures, each of which represents a format-dependent manager. Each structure contains a set of pointers to routines that implement the manager.

8.5.1 Format-Dependent Routines

Each format-dependent manager implements a set of routines that includes the following (this is a partial list):

recog( )
Examines an object module to determine whether it is in a format supported by this format-dependent manager. The loader determines which format-dependent manager to use by calling the installed recognizer routines one by one until one of them recognizes the object module being loaded.

get_imports( )
Constructs the import symbol table and import package table for a module.

map_region( )
Maps the regions of an object file into the process's address space.

get_export_pkgs( )
Returns the list of packages exported by this object module.

get_exports( )
Returns the list of exported symbols for the specified object module. This routine is not called by the format-independent manager in normal module loading. It is intended for use only when preloading modules, and possibly to allow format-dependent managers such as ELF to implement their own symbol resolution algorithms.

lookup_export( )
Used during symbol resolution to locate the specified package name/symbol name pair in the specified object module.

relocate( ) Cycles through the relocation records, relocating all relocatable addresses in the module. The routine uses the region array built in the **map_regions( )** call, and the imported symbols and import packages arrays built in the **get_imports( )** call.

get_entry_pt( )
Returns the address of the module's entry point, if one exists.

8.6 Address Space Management

The OSF/1 loader address space management design meets the following goals:

- Address space management is inherently machine-dependent, but does not need to be format-dependent.
- Address space configuration (where the code, data, stack, shared libraries, loader, and so on are mapped by default) should be maintained in only one place because it is machine-dependent information.
- The format-dependent loader routines (and the core format-independent loader routines) should be independent of the loader context.

The format managers use allocation and deallocation procedure interfaces to decide where to map a given region and to deallocate any allocated space during cleanup or unmap. Although the loader maps regions using format-dependent routines, those routines uses a format-independent interface to assign the addresses to a region being mapped.

8.6.1 Absolute and Relocatable Regions

Regions in an object file can be classified as *absolute* or *relocatable*. The loader must load absolute regions at a fixed address specified by the object file. The loader can load relocatable regions at any address. Shared libraries and dynamically loaded modules are usually relocatable.

8.6.2 Base Addresses and Virtual Addresses for a Region

There is typically a distinction in the loader's memory allocation interfaces between a *virtual address* for the region, which is where it will be loaded in the target process, and a *base address*, which is the address it occupies in the current process. This distinction is critical for relocatable regions because they must be relocated to the virtual address, even if they are mapped somewhere else.

If the base address and the virtual address are the same, the region is absolute—it must be loaded at a specific location in memory. If the base address and the virtual address are not the same, then the region may be mapped anywhere in the current address space because it will eventually be mapped and run elsewhere.

8.6.3 Context-Specific Allocation Procedures

A special allocator is necessary for the kernel context because the base address for mapping regions is almost always different from the virtual address at which the region will reside. This difference exists because the base address is in the address space of the kernel load server and the virtual address is in the address space of the kernel.

The preload context requires special allocation procedures because the space must be allocated from a range of memory that is especially reserved for prerelocated libraries, so that all processes can map the prerelocated regions at the same addresses. As with the kernel context, preload libraries have base addresses different from their virtual addresses. They are simply loaded into **lib_admin**, and then copied into the preload file.

8.6.4 Typical Loader Address Space Usage

The region allocation procedures for the process context and the preload context use an address space configuration record that is machine-dependent and resides in the kernel. This record is read from the kernel by the **getaddressconf( )** system call. It contains, among other things, the base address, growth direction, and flags of each of the following areas:

- Program *text* area. This is the address of the default text area, where absolute code is linked to run.
- Program *data* area. This is the address of the default data area, where absolute data is linked to run.
- Program *bss* area. This is the address of the default *bss* area, where absolute code is linked to run.
- Stack area.
- Loader text area. This is the absolute address the loader itself is linked to run.
- Loader data area.
- Loader bss area.
- Loader private data file (inherited).
- Loader global data file (IPT and heap).
- Loader preloaded library data.
- **mmap** ed file text.
- **mmap** ed file data.
- **mmap** ed file bss.

The program text, data, and bss areas are the addresses of the default text, data, and bss areas where absolute code is linked to run. The loader text area is the absolute address where the loader itself is linked to run. Libraries that are not preloaded go into the **mmap** areas, respectively, for their text, data, and bss.

8.7 Kernel Space Loading

OSF/1 allows modules to be loaded into the kernel at runtime, so that adding new device drivers and network protocols does not require shutting down and rebooting the system. As described in Chapter 9, system administrators can use the **sysconfig** command to dynamically configure the kernel. This command invokes the system's configuration manager, which manages the loading and configuration of dynamically loaded modules.

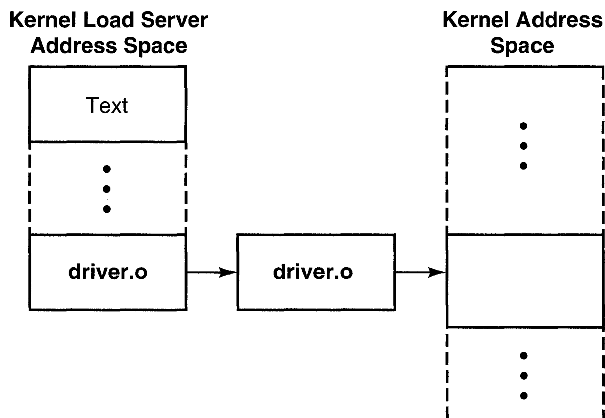
The operation of loading such modules into the kernel is performed by a privileged user space process called the *kernel load server*. The server runs in user space because it needs to call the user space loader. It is a separate process because it needs to maintain the state information (that is, modules, exports lists, and so on) of what has been loaded into the kernel. The kernel load server is privileged because it can modify the kernel's address space.

Loading a module into the kernel requires several steps:

1. The kernel load server calls the loader and specifies the kernel context. The kernel context specifies which region allocation routines are to be used.
2. The module's regions are mapped into the kernel load server's address space.
3. The regions are relocated to the kernel's address space.

Figure 8-5 illustrates this scheme.

Figure 8–5. Kernel Load Relocation



8.8 Preloading, Installing Libraries, and the Global Data File

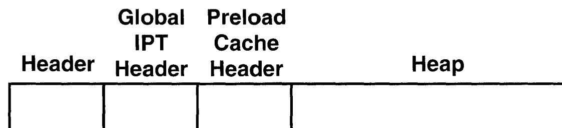
To support efficient library sharing, OSF/1 uses the loader to *preload* or *prerelocate* libraries. A library administration tool, **lib_admin**, installs and preloads libraries into a global data file and maintains a *preload cache* of completely loaded shared libraries, already relocated to runtime addresses with all symbols fully resolved. This cache, as well as the *global installed package table*, is maintained in a system file. The loader accesses preloaded libraries by mapping them into the target address space at their predetermined runtime addresses.

A program that uses a prerelocated shared library has typically been compiled in such a way that its text section is position-independent. This means that the loader does not need to apply any relocations to the text section when loading the program for execution. If any relocation is necessary to the main program to resolve share library addresses, the addresses will most likely be in the data section of the program.

Note that preloading is not a specific loader function. The preload cache is managed by the **lib_admin** command, which loads the shared libraries into a specially created context, and then copies them into the preload cache.

The system file that holds the preload cache has the layout shown in Figure 8-6.

Figure 8–6. Layout of the Preload Cache Data File



The global IPT records reside in the heap area of this file. The preload cache stores a full set of information (module records, region records, export lists, and module name hash tables) for the set of preloaded libraries currently available.

Prerelocation of shared libraries is based on the fact that every process can map the prerelocated regions at the same addresses. Therefore, a range of memory is reserved for prerelocated libraries. If a process allocates memory in the reserved area for other purposes, it may not be able to use the prerelocated versions of some libraries.

8.9 Dynamic Format Manager Loading

The loader supports both built-in format-dependent managers and dynamically loadable auxiliary format-dependent managers. When the built-in managers fail to recognize a module being loaded, the loader attempts to load a dynamic manager. The dynamic managers are simply listed in an ASCII text file; they are loaded one by one until the module is recognized.

Dynamic managers are loaded into their own loader context to reduce naming conflicts and avoid problems with using the same manager in multiple contexts. The dynamic manager context is created when the first dynamic manager is loaded. It is bootstrapped to contain all the symbols of the format-independent loader, including loader utilities and so forth.

There are many restrictions on dynamic managers:

- They may not have export symbols.
- They may only use symbols exported by the format-independent manager.
- They must be in a recognized format.

8.10 Unloading

The loader supports a simple unload; it unmaps the module's regions and discards the loader data structures describing the module. References to a module may become invalid once the module is unloaded. The loader does not keep track of references or attempt to unsnap such invalid links. These housekeeping tasks are the responsibility of the process doing the unload.

8.11 Application Interface to the Loader

The loader provides an application programming interface that allows programs to dynamically load and unload modules from their address spaces. The **load( )** system call allows a running process to load modules into its address space. A dynamically loaded module can have imported symbols that resolve either to other modules, to previously loaded shared libraries, or to shared libraries that have not yet been loaded.

The **unload** system call can be used to unload a module. This call does not attempt to deal with references to the unloaded module; it is the application's responsibility to prevent references to an unloaded module.

8.12 The Loader and Security

A privileged program (**setuid**, **setgid**, or executing with any privilege bits set in its privilege vector) will always be loaded with the default loader. This ensures that clever users cannot use their own loaders to load arbitrary privileged programs.

The loader design also protects the user of shared libraries. The loader finds its shared libraries by consulting the installed package tables. The global installed package table is stored in a portion of the file system that is writable only by root.

A process inherits its private installed package table from its parent through a memory segment that is mapped with the **keep-on-exec** bit turned on. When **exec()** loads a privileged process, it deallocates all regions, even those that are marked **keep-on-exec**. This prevents privileged processes from inheriting private libraries that might be used to breach the system's security.

See Chapter 15 for more information on the OSF/1 security architecture.

Chapter 9

Loading and Configuring Dynamic Subsystems

Kernel subsystems are components of the operating system whose functions are logically separate from functions of the core kernel. File systems, network protocols, and device drivers are examples of kernel subsystems.

In traditional UNIX systems, kernel subsystems are linked directly into the kernel at build time. On those systems, adding a new subsystem to the kernel requires that the kernel be recompiled and linked, and the operating system shut down and rebooted. The OSF/1 kernel supports the dynamic loading and configuring of subsystems. Modules for new device drivers, file systems, network protocols, and system calls can be added while the operating system is running, without having to rebuild the kernel, shut down, and reboot.

OSF/1 supports the dynamic loading and configuration of the following types of subsystems:

- Block and character device drivers, including pseudo-device drivers
- File systems
- Socket-based network protocol families
- STREAMS modules and drivers

Subsystems can also be dynamically unloaded and unconfigured.

9.1 Overview: Loading and Configuring Dynamic Subsystems

The system administrator maintains a configuration database that describes the various subsystems. To add a new subsystem to the kernel, the administrator updates the database to include information about the subsystem; for example, the pathname of the subsystem's object module.

The administrator then executes the **sysconfig** operator command to load and configure the subsystem. This command issues a request to the system's configuration manager daemon, which uses the information in the configuration database to load and configure the subsystem.

The configuration manager daemon loads the subsystem's module into the kernel's address space by invoking the kernel load server. When the module is fully loaded, the configuration manager daemon then issues a system call that causes control to be turned over to the module's configuration routine. This routine performs the operations required to configure the module.

9.2 Configuration and Kernel Tables

The kernel references its subsystems through a set of tables. For example, all block device drivers are referenced through the kernel's block device table, all character device drivers are referenced through the character device table, and so on. These tables can be modified when the system is active.

When a subsystem is dynamically configured, it uses a set of system calls, based on the subsystem's type, to register itself in the appropriate kernel tables. For example, a device driver registers itself in the interrupt vector table and the appropriate device switch tables, a file system registers itself in the VFS switch table, a network protocol family registers itself in the protocol family table, and so on.

A dynamic subsystem's configuration routine is responsible for registering the subsystem in the appropriate tables. A module never modifies any of the configuration tables directly; instead, it calls a service routine in the kernel

that performs the operation. For example, the service routine **domain_add()** registers a network protocol family in the network protocol family table.

The kernel services provide a clean separation between the kernel proper and the subsystems. For each subsystem type, the kernel defines a set of data structures and interfaces that allow subsystems to "hook" themselves into the kernel. These interfaces and structures are called *frameworks*.

9.3 The Configuration Manager

The configuration manager handles all requests for configuration, unconfiguration, reconfiguration, and querying of subsystem modules. The requests fall into two categories:

- Configuration command requests through interprocess communication (for example, the system administrator's requests through the **sysconfig** command), which occur during system operation
- Automatic configuration requests made during startup

Several steps are required for dynamic configuration of a subsystem:

1. The configuration manager reads the configuration database in the file **/etc/sysconfigtab** to get information about the subsystem to be loaded and configured. The database contains information about the subsystem, including its description, method and type, and the location of its object module. The configuration method is the subsystem-specific part of the configuration manager that runs in user space. It consists of a set of functions to handle the subsystem-dependent entries in the configuration database.
2. The configuration manager then calls the kernel loader, **kloadsrv**, to load each subsystem's object module into the kernel. Upon success, **kloadsrv** returns the subsystem's ID and entry point, which the configuration manager stores into a registration table. The entry point is the address of the routine the subsystem uses to configure itself.
3. The configuration manager then calls the **kmodcall()** system call, which looks in the registration table to find the configuration entry point for the subsystem. The **kmodcall()** system call executes the

entry point with the configure option. The subsystem's configuration routine then configures the subsystem into the kernel. Any specific subsystem type information is passed back to the configuration manager. If the subsystem configuration fails, then the subsystem module is unloaded.

9.4 Interrupt Handling

The interrupt handling scheme in OSF/1 supports the loading and unloading of device drivers while the kernel is running. The interrupt handling mechanisms are supported by a consistent, modular strategy in which device-dependent code is separated from device-independent code.

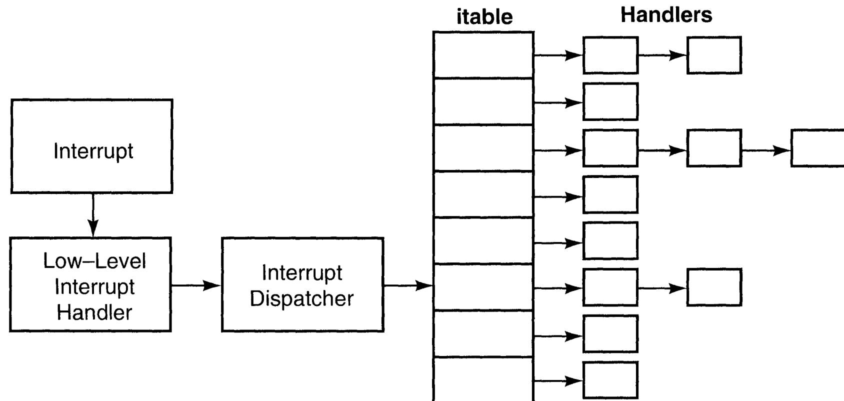
The interrupt handling model is divided into three distinct parts:

- An assembly language module, **locore.s**, includes a low-memory data structure holding the addresses of entry points to execute when an interrupt is received. It also includes the low-level interrupt handlers. There is normally a low-level interrupt handler for each hardware interrupt supported.
- The interrupt dispatcher is the code that traverses an array that points to an arbitrary number of registered interrupt handlers.
- The device interrupt handlers are loaded into the kernel when the device driver is loaded and are registered with the interrupt dispatcher when the device driver configures itself.

9.4.1 The locore.s Module

The **locore.s** module declares low memory and includes a set of low-level interrupt handlers to service the various CPU interrupts (see Figure 9-1). When an interrupt is received, the low-level interrupt handlers perform the necessary functions to invoke the interrupt dispatcher. The use of an interrupt dispatcher contrasts with most traditional schemes, which call the device interrupt handlers directly from **locore.s**, sometimes through "glue" code.

Figure 9–1. Interrupt Handling



9.4.2 The Interrupt Dispatcher

The interrupt dispatcher is the body of code that actually invokes each device interrupt handler. The interrupt dispatcher, when called as a result of a hardware interrupt, reviews the interrupt handler table, **itable**, for all registered and enabled interrupt handlers at the interrupt level passed to it. If the dispatcher finds an interrupt handler for that level, it invokes it.

The algorithm for performing the lookup into **itable** is left unspecified and may be influenced by the hardware architecture. The interrupt handler table, **itable**, consists of an array of pointers to interrupt handler structures. The actual details can vary according to the underlying hardware architecture.

The interrupt dispatcher code does not need to reside in **locore.s**, nor does it require the traditional "glue" code found in **locore.s**. By introducing the interrupt dispatcher and supporting data structures, the system has a great deal of flexibility that was not generally available with previous mechanisms.

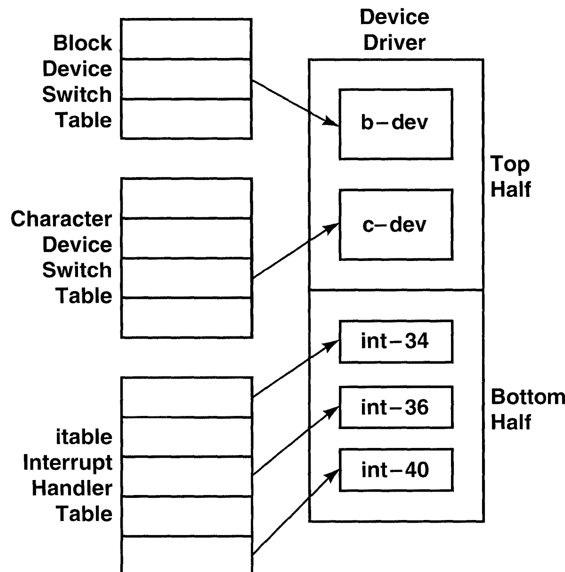
9.5 Device Driver Configuration

Figure 9-2 shows kernel table entries pointing to the several parts of a configured device driver. The top half of the driver has both a block device and a character device. An entry in the block device table switch points to the block device handler in the top half. Likewise, an entry in the character device table switch points to the character device handler in the top half. The bottom half of the driver has handlers for three separate hardware interrupts. (The names **int34**, **int36**, and **int40** are arbitrary.) Three separate entries in the interrupt handler table (**itable**) point to the three interrupt handlers.

To configure a device driver, the kernel needs to register the interrupt handlers of the driver's bottom half into the **itable**, and to register the system call handlers of the top half into the block and character device switch tables. Additionally, the interrupt handlers, once registered, must be enabled.

Each dynamically configurable device driver must supply the kernel with the entry point of a self-configuration routine. The routine is responsible for proper initialization of the device, for calling the kernel services that register the handlers of the top and bottom halves, and for calling the kernel services that enable the interrupt handlers.

Figure 9–2. Device Driver Configured into Kernel Tables



As new device drivers are added to a running system, their interrupt handling entry points are dynamically registered in **itable**. The registration process occurs when the device driver's configuration entry point is called at configuration time. The device driver issues the **handler_add( )** call, passing a filled-in interrupt handler structure (**ihandler_t**) and receiving back an interrupt handle (**ihandler_id_t**). The device driver then needs to enable its interrupt routine by calling **handler_enable( )**, passing the handle just received from **handler_add( )**. The OSF/1 interrupt handling scheme provides the necessary interfaces to enable or disable interrupt handlers that have already been loaded. Once an interrupt handler has been enabled, it is available to service interrupts and, therefore, is an entry point that the dispatcher can execute.

Pseudodevices, such as the traditional null device (**/dev/null**) and the pseudoterminals, have no physical device, and thus no hardware interrupts and no bottom half. Only the top half needs to be configured. The driver's self-configuration routine registers the top half. The self-configuration routine of the driver for a physical device must register both the top and bottom halves.

9.6 Configuration of File Systems

OSF/1 contains a Virtual File System (VFS) framework that supports multiple file system types. File system types can be either statically configured into the kernel or made dynamically loadable. Dynamically loaded file system types can also be unloaded when they are idle, but statically loaded file system types cannot. Chapter 11 provides a complete description of the VFS framework.

The VFS framework contains a table of configured file system types called the *VFS table* or the *VFS switch* (**vfssw**). This table contains a pointer to the file system-specific operations that provide file system-level functions, such as **mount()**, **umount()**, **statfs()**, and **sync()**. File systems that have been loaded into the kernel statically are already configured into the **vfssw**, and are initialized at boot time through calls to their VFS initialization function, **vfsinit()**.

Once a file system type has been loaded into the kernel, a call to its configuration entry point causes it to configure itself into the **vfssw**, through the **vfssw_add()** kernel service routine, and to perform appropriate initialization of global parameters, allocation of memory, and so forth.

Dynamically unloading a file system type is almost the reverse of loading. Once it has been determined that the file system is not in use, its entry in the **vfssw** can be deleted, and its text unloaded from the kernel address space.

The actions of unloading are similar to those of loading. The file system's configuration entry point is called for unconfiguration, which results in the configuration routine calling **vfssw_del()**. If any instance of the file system type is mounted, **vfssw_del()** fails. If **vfssw_del()** succeeds, the file system has been unconfigured, and the configuration routine needs only to clean up after itself and to return. (An example of cleanup is deallocation of system memory.) The kernel loader process then unloads the file system's text from the kernel address space.

9.7 Dynamic Loading and Configuring of System Calls

New system calls can be dynamically added to a running copy of OSF/1. The newly configured system calls are immediately available to executing user-level programs that have been properly constructed to use them.

A user-space program can invoke dynamically loaded system calls by name, in the same manner that it would invoke other system calls; there is no need for the application programmer to know that they are dynamically loaded.

Each system call in the OSF/1 kernel, whether built-in or dynamic, consists of a body of executable code that is identified by a unique name and a unique number. Dynamic loading places a new system call's body of code into kernel space; dynamic configuration associates the name and the number with the code through appropriate changes to the kernel's system call table.

9.7.1 Selecting the System Call Number

The provider of a subsystem may use either a predetermined or dynamic approach for determining the system call numbers associated with its exported system calls.

With the predetermined approach, the provider assigns the numbers and ensures that **nosys** entries occupy the appropriate places in the **syscalls.master** file (in **kernel/conf**) to reserve slots in the system call table at build time. Upon configuration, the subsystem provides the number to the kernel. There is only one significant difference between this method and the original static system calls: it is not necessary for the symbol that is the target of the system call to be defined at build time.

In the dynamic approach, the numbers are provided by the system at load time, using unassigned slots in the system call table. This method assumes enough free slots remain in **syscalls.master**, because no mechanism is provided to increase the actual size of the system call table.

The predetermined approach provides slightly better performance and less complexity in the user-space code; the dynamic approach allows greater flexibility and avoids the necessity of knowing about specific dynamic

system calls when the kernel is built. Only the predetermined approach is appropriate for a secure system that audits system call usage.

9.8 Boot-Time Subsystem Configuration

There are two separate stages to OSF/1's subsystem configuration at boot time. The kernel handles the first stage, determining the sizes of several fundamental parameters, initializing statically bound device drivers, and starting the **init** process. The **init** process starts several other processes, including the kernel loader and the configuration manager. The configuration manager then handles the dynamic configuration, which is the second stage.

During the system generation process, the software performing the build consults the static system configuration file to determine which device drivers to statically bind into the kernel and at which location (or locations) the corresponding physical devices may be present.

Device drivers may thus be statically linked and loaded into the kernel in the usual manner. In the course of static configuration at boot time, after the kernel has determined the sizes of several fundamental data areas and other parameters, it calls the static drivers' entry points so that they may perform any necessary initialization. They may, for example, probe devices, add switches, or install system calls. Probing may, in the usual manner, result in configuration of only those drivers for which the physical devices are detected.

Chapter 10

Internationalization Subsystem

Traditionally, application programs developed on UNIX systems were written with a bias toward the English language and the customs of the United States. Such applications provide error messages that are written in English and display numeric data, monetary data, and time and date data according to the cultural conventions of the United States.

When application programs of this type are used outside of the USA, users encounter difficulties such as

- Messages in an unfamiliar language
- Incorrect alphabetic sorting
- Unfamiliar date, numeric, and monetary displays

The internationalization subsystem supports internationalized applications. An internationalized application is one that is capable of behaving properly regardless of a user's language and cultural conventions. For example, an internationalized application can be used successfully by users in the USA, Europe, and Japan.

10.1 Locales

OSF/1 implements internationalization support through a set of *locales*. Each locale specifies a software environment that supports the language and customs associated with a particular geographic region. A locale specifies the following:

- A language and a code set that will be used to represent the language. For example, American English is represented by the ASCII code set.
- Collating conventions.
- Format conventions for the display of times, dates, numeric data, and monetary data.
- A catalog of messages the application uses to communicate with the user.

An application determines the current locale at runtime, usually by means of the user's environment variables. The application then uses the **setlocale()** routine to give itself access to the tables and algorithms that implement the locale. When the application performs an operation that is locale-dependent, the routine that performs the operation uses the algorithm and data specific to the locale.

10.1.1 Languages and Code Sets

In an internationalized application, users interact with the system in their native language. All program messages are in the local language, and the program accepts input in that language. Instead of being hardcoded into the program, messages are placed in *message catalogs*, and hardcoded text is replaced with calls to a messaging system. To specify postive or negative responses, users can use the words or characters appropriate to their language instead of the English string literals **y**, **yes**, **n**, and **no**.

Traditionally, character data manipulated by UNIX applications has been represented by the ASCII code set, which is capable of representing all the characters for only three languages: English, Hawaiian, and Swahili. To support other languages, an application must be capable of using code sets that represent those languages.

The ASCII code set uses seven bits of each byte and cannot encode non-English characters. To allow additional characters, other code sets either use all eight bits in a byte or use multiple bytes to encode a character. Eight-bit code sets allow 256 possible characters and can support European, Middle Eastern, and other alphabetic languages.

In languages that use ideographic writing systems, such as Japanese, Chinese, and Korean, each word has its own unique ideographic symbol or symbols. There are thousands of such symbols in these languages. Consequently, these languages cannot be coded within a single byte and require multiple bytes for most characters. Multibyte encoding methods combine both single-byte and multibyte code sets.

In addition, universal character sets have been designed that include characters from a large group of languages. These universal sets can consume from two to four bytes per character.

One distinguishing characteristic of multibyte code sets is that their characters can have different lengths. For example, the SJIS code set, which allows the ASCII code set to be combined with the standard 16-bit Japanese code set called JIS X0208, includes 1-byte and 2-byte length characters. Code sets with characters that vary in length can introduce inefficiencies in applications that manipulate data a character at a time because such an application must check the length of each character before processing it.

The internationalization subsystem includes a set of interfaces that allows applications to convert variable length characters into *wide* characters of a uniform length for the character manipulation. Characters that have been converted into wide length characters are in a form called *process code*.

The size of wide length characters is system-dependent; a wide length character may be 16 bits on one system and 32 bits on another. Consequently, characters encoded in process code cannot reliably be transmitted between processes on different systems. Therefore, before an application performs an I/O operation involving process code, it must translate the code back to the original multibyte form.

The following interfaces can be used to convert characters to and from their multibyte forms:

mbtowc( ) Converts a multibyte character to a wide character

mbstowcs( ) Converts a multibyte character string to a wide character string

wctomb( ) Converts a wide character to a multibyte character

wcstombs( ) Converts a wide character string to a multibyte character

mblen( ) Returns the number of bytes in a multibyte character

It is possible for a single code set to handle more than one language. For example, the same code set can be used for Western European languages such as French, German, Italian, and Spanish. It is also possible to have more than one encoding method for a single language. OSF/1 includes support for two different encoding methods for Japanese. See Section 10.5 for further discussion.

10.1.2 Collating Conventions

English sorting rules are among the simplest of any language: each letter sorts to one place. ASCII makes things even simpler by encoding the characters in alphabetic, case-segregated order. Other languages include a variety of collation methods. Here are a few examples:

Multilevel
: In this system, a group of characters all sort to the same primary location. If there is a tie, a secondary sort is applied. For example, in French, **a**, **á**, **à**, and **â** all sort to the same primary location. If two strings collate to the same primary location, the secondary sort goes into effect. These words are in correct French order:

  a
  à
  abord
  âpre
  après
  âpreté
  azur

One-to-two character mappings
: This system requires that certain single characters be treated as if they were two. For example, in German, **ß** (Eszett) is collated as if it were **ss**.

N-to-one character mappings

Some languages treat a string of characters as if it were one single collating element. For example, in Spanish, the **ch** and **ll** sequences are treated as their own elements within the alphabet. Dictionaries have separate sections for them (that is, there are entries for **a**, **b**, **c**, **ch**, **d**, and so on). The following words are in correct Spanish order:

canto
construir
curioso
chapa
chocolate
dama

Don't-care character mappings

In some cases, certain characters may be ignored in collation. For example, if a **-** (dash) were defined as a don't-care character, the strings re-locate and relocate would sort to the same place.

In addition to these collation rules, some languages use basically the same rules as English but still need more than a plain ASCII sort. For example, in Danish, there are three characters that appear after **z** in the alphabet: **æ**, **ø**, and **å**. This means that an internationalized application cannot assume that the range [A-Z, a-z] includes every letter.

A locale may include tables that specify the operations used by applications to collate characters, compare characters, and perform regular expression operations within the locale. An application accesses these operations through the following function calls:

strcoll( ) Collates two multibyte strings based on the locale's collation tables

strxfrm( ) Converts a multibyte string into a form that collates correctly, according to the locale's collation table, when collated by **strcmp( )**

wcscoll( ) Collates two wide character strings based on the locale's collation tables

wcsxfrm( ) Converts a wide character string into a form that collates correctly, according to the locale's collation table, when collated by **wcscmp( )**

fnmatch( ) Matches filename patterns

regcomp( ) Compiles regular expressions for later comparisons

regerror( ) Returns text associated with an error code from **regcomp( )** or **regexec( )**

regexec( ) Compares a string to a compiled regular expression

10.1.3 Character Classification

The new characters that are necessary to support languages besides English need classification. For European languages, the existing classes, such as **alpha** and **lower**, are adequate. The additional characters that are valid for a given language and class need to be provided. In addition, some characters have qualities that do not exist in the ASCII code set. For example, the German **ß** is a lowercase letter that has no single uppercase equivalent. Therefore, **islower( )** would return TRUE on this letter, while **toupper( )** would return the original character (**ß**).

10.1.4 International Date and Time Formats

Users around the world express dates and times using a variety of formatting conventions. When specifying day and month names, Americans generally use this format:

Tue, May 22, 1990

However, the French use this format:

mardi, 22 mai 1990

An internationalized system gives users access to their language's conventions.

Cultural groups also express numeric dates in different ways, even within a single country. The following examples illustrate common methods for formatting dates:

3/20/90	American: month/day/year order
20/3/90	British: day/month/year order
20.3.90	French: day.month.year order
20-III-90	Italian: day-month-year order; uses the Roman numeral for the month
90/3/20	Japanese: year/month/day order
2/3/20	Japanese Emperor: same order, but the year is the number of years the current emperor has been reigning, rather than the Gregorian calendar year

As with dates, there are many conventions for expressing the time of day. Americans use the 12-hour clock with its a.m. and p.m. designations, while most people in Europe and Asia use the 24-hour clock for written times.

In addition to the 12-hour/24-hour clock differences, punctuation for written times can vary. For example:

3:20 p.m.	American
15h20	French
15.20	German
15:20	Japanese

With different date and time formats come different time zones, which can vary in one-hour, 30-minute, or even 15-minute increments.

A locale may include tables that specify the format of time and date data for the locale. An application uses the following interfaces to format time and date according to the locale:

strftime( )	Converts a date and time value to a string
strptime( )	Converts a string to a date and time value
wcsftime( )	Converts a date and time value to a wide character string

10.1.5 International Numeric and Monetary Formats

The characters used to format numeric and monetary values vary from place to place. For example, Americans use a . (period) as the radix character (that is, the character that separates whole and fractional quantities), and a , (comma) as a thousands separator. In many European countries, these definitions are reversed. In addition, for monetary amounts, there are a variety of conventions for the currency symbol and its placement. For example:

Numeric Formats

1,234.56	American - comma as thousands separator; period as radix character
1.234,56	French - period as thousands separator; comma as radix character

Monetary Formats

$1,234.56	American dollars
kr1.234,56	Norwegian krona
SFrs.1,234.56	Swiss francs
1.234$56	Portuguese escudos

10.2 Internationalization Subsystem Design

The OSF/1 internationalization subsystem allows applications to behave differently in different locales by dynamically loading the code and tables implementing the user's locale at application runtime.

The requirements of the internationalization subsystem design and the benefits of object-oriented programming coincide. Defining locales and related items as objects and providing methods to access these objects creates clearly defined interfaces and highly modular components that can be substituted for each other when running an application program.

The overall subsystem design consists of three parts:

- Object-oriented framework
- Object definitions that provide the application programming interface (API)
- Object definitions that specify the algorithms and data structures

The object-oriented framework consists of the rules used to define the subsystem's objects and the mechanisms used to implement them. The framework provides a mechanism for implementing the internationalization API, but is not tied to it. It is possible to implement an entirely different API using the same object-oriented framework.

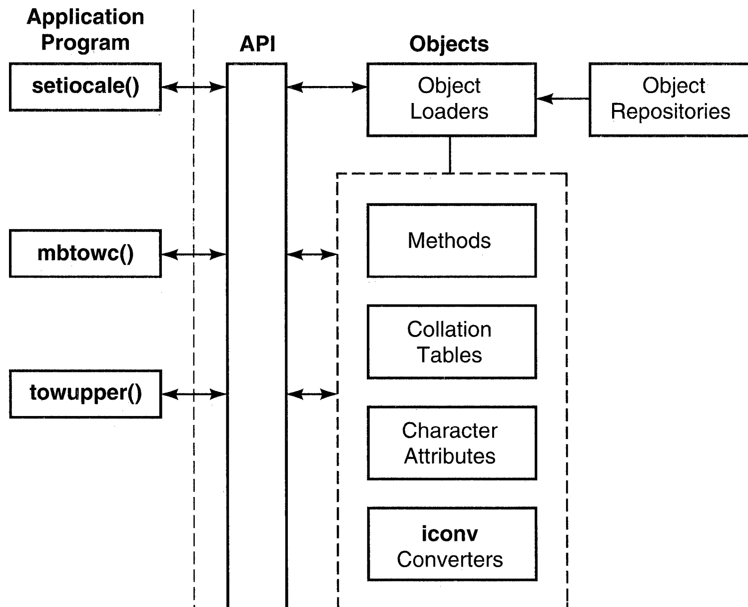
The objects defined in the framework provide the internationalization API. This part of the design specifies the relationship between the internationalization interfaces and the objects in the subsystem. It specifies which interfaces are included in each object and how interfaces access the objects.

Within the objects are methods and data structures. This part of the design specifies the algorithms used within the methods to manipulate the data.

10.3 Application Programming Interface

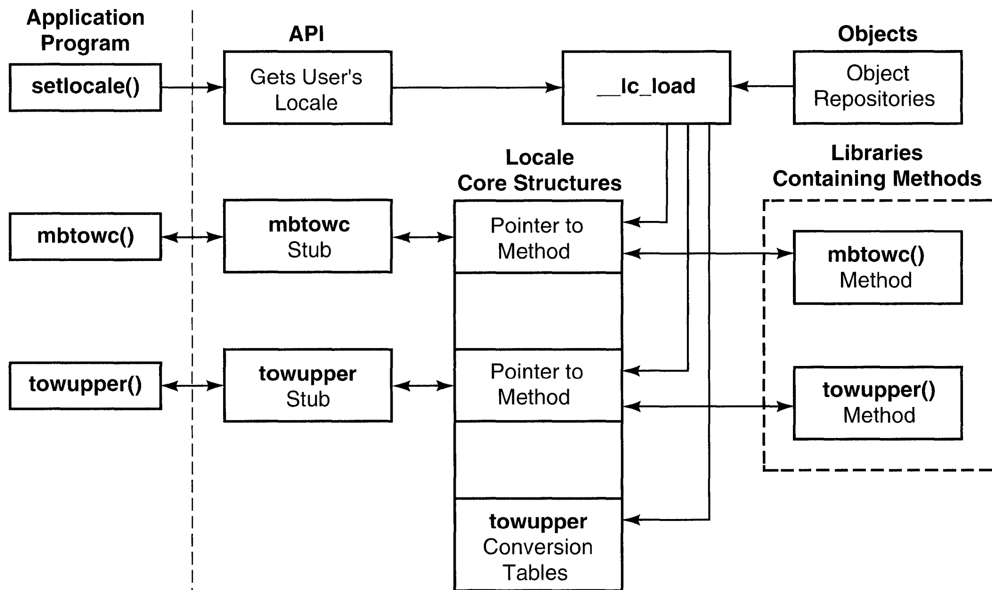
Although the internationalization subsystem has an object-oriented design, the objects are not visible to the applications which use the subsystem. The objects and methods are only used internally to provide the API. This section describes the connections between the API and the object-oriented subsystem. Figure 10-1 illustrates how an application program interfaces with the object-oriented internal subsystem.

Figure 10–1. Internationalization Subsystem Application Programming Interface



This object-oriented design is used for the **setlocale**( ) related functions and for the **iconv**( ) converters. Figure 10-2 provides more details on how the object-oriented design works. It illustrates how the **setlocale**( ) function loads the locale core objects from the repository and how the **mbtowc**( ) and **towupper**( ) functions access the locale core structures to execute their functions. The **setlocale**( ) function evaluates the internationalization environment variables to get the user's locale, then uses **__lc_load**( ) to load the objects from the object repository.

Figure 10–2. Internationalization Objects



10.4 Message Subsystem

The message subsystem implemented in OSF/1 does not use the object-oriented paradigm. The API for the message subsystem consists of the following function calls:

catopen( ) Opens a version of a named message catalog as determined by the current locale

catgets( ) Retrieves a specific message string from the message catalog

catclose( ) Closes the specified catalog

The **catopen( )** function evaluates the environment variables and the locale specified by **setlocale( )**, and then stores this information with the catalog name in the **catdtbl** table. It does not attempt to open or even find the

catalog. The catalog is actually opened by the first call to **catgets( )**. Deferring the catalog open until a message is needed can improve the efficiency of applications. Application startup is faster because the catalog does not have to be opened. If the application program does not encounter any conditions that cause a message to be displayed, the overhead of the catalog open is eliminated entirely.

The **catgets( )** function first tests if the catalog has been opened. If the catalog has not been opened, it calls the internal interface **_cat_do_open( )**, which loads the catalog using the information in the **catdtbl** table and the **mmap( )** system call.

Using **mmap( )** to access message catalogs has several advantages:

- Catalogs are shared between users. The OSF/1 VM optimization for caching and sharing memory can be used.
- No input/output descriptor is retained. This simplifies using catalogs in programs (such as shells) that attempt to manage their input/output descriptors.
- Message access is fast because file input/output is avoided in favor of VM page faults.

After the file has been opened, **catgets( )** uses a table of offsets stored in the message catalog to get a pointer to the message using the *set_number* and the *message_number* parameters specified in the call to **catgets( )**.

The **catclose( )** function cleans up internal storage so that a subsequent **catopen( )** can specify a new catalog, but if the same catalog is used again in the process, the previously loaded copy is reused.

10.5 OSF/1 Code Sets

OSF/1 accepts data encoded in the series of 8-bit code sets defined by ISO 8859. The first in the series is called ISO 8859-1, the second is ISO 8859-2, and so on through ISO 8859-9. Although OSF/1 accepts data encoded in any of the ISO 8859 series, it provides locales for some languages in the ISO 8859-1, ISO 8859-7, and ISO 8859-9 code sets only.

ISO 8859-1 is often called Latin-1. It includes the characters necessary for Western European languages, such as French, German, Italian, and Spanish. Latin-1 and the other ISO 8859 code sets are arranged so that they include ASCII characters at their traditional 0x0 through 0x7f code positions, control characters at positions 0x80 through 0x9f, and additional graphic characters at positions 0xa0 through 0xff.

Table 10-1 lists the ISO 8859 code sets.

Table 10–1. ISO 8859 Code Sets

Formal Name	Informal Name	Languages Covered
ISO 8859-1	Latin-1	Western European
ISO 8859-2	Latin-2	Eastern European
ISO 8859-3	Latin-3	Southeastern European
ISO 8859-4	Latin-4	Northern European
ISO 8859-5		English and Cyrillic-Based
ISO 8859-6		English and Arabic
ISO 8859-7		English and Greek
ISO 8859-8		English and Hebrew
ISO 8859-9		Western European and Turkish

Latin-1 includes Danish, Dutch, English, Faeroese, Finnish, French, German, Icelandic, Italian, Norwegian, Portuguese, Spanish, and Swedish.

Latin-2 includes Albanian, Czech, English, German, Hungarian, Polish, Rumanian, Serbo-Croatian, Slovak, and Slovene.

Latin-3 includes Afrikaans, Catalan, Dutch, English, Esperanto, German, Italian, Maltese, Spanish, and Turkish.

Latin-4 includes Danish, Estonian, English, Finnish, German, Greenlandic, Lappish, Latvian, Lithuanian, Norwegian, and Swedish.

10.5.1 EUC Code Sets

OSF/1 supports the EUC (Extended UNIX Codes) encoding methods for encoding Japanese and other ideographic languages and the SJIS (Shifted Japanese Industrial Standard) code set for encoding Japanese.

EUC is an encoding standard that allows several code sets to be combined. The first byte of an EUC character determines the code set, the number of bytes to encode the character, and the display width of the character. Table 10-2 illustrates the OSF/1 Japanese EUC implementation.

Table 10–2. OSF/1 Japanese EUC Code Set Encoding

Character Type	Character Set	Value of First Byte	Total Number of Bytes	Display Type
ASCII	0	0x00—0x7F	1	1
Kanji	1	0xA1—0FE	2	2
Kana	2	SS2 (0x8E)	2	1
Kanji	3	SS3 (0x8F)	3	2

In the Japanese EUC multibyte code sets, the bytes following the first one are always in the range 0xA1 through 0xFE.

10.5.2 SJIS Code Set

SJIS allows ASCII to be combined with a standard 16-bit Japanese code set called JIS X0208. The characteristics associated with a particular value vary from implementation to implementation. Typically, if the byte has the Most-Significant Bit (MSB) set to 1, and its value is between either 0x81 and 0x9f, or between 0xe0 and 0xfc, the byte is the first of a 2-byte character. Any character sequence that does not begin with one of the special "first of two" bytes is treated as a 1-byte character. If the MSB is off, that 1-byte character is ASCII; if it is on and in the range of 0xa1 through 0xdf, the character is a single-byte phonetic character. Table 10-3 describes the OSF/1 SJIS encoding method.

Table 10–3. OSF/1 SJIS Encoding Method

Character Type	Value of First Byte	Total Number of Bytes
ASCII	0x00-0x7F	1
Kana	0xA1-0xDF	1
Kanji	0x81-0x9F or 0xE0-0xFC	2

In the OSF/1 implementation, the second byte is always in the range 0x40 through 0xFC.

10.6 The iconv Conversion Subsystem

The **iconv** conversion subsystem converts data encoded in one code set to data coded in another. Since the code sets it is converting from and to are independent of the current locale, the **iconv** conversion subsystem is independent of the locale objects used in the internationalization subsystem.

The API for the **iconv** conversion subsystem consists of the following functions:

iconv_open( ) Initializes the code set converter

iconv( ) Converts the specified data

iconv_close( ) Closes the code set converter

In order for **iconv** to be able to convert from one code set to another, there must be present either a method explicitly defined for the conversion or a conversion table explicitly defined for the conversion. The **iconv_open( )** function uses the code set names specified in its parameters and the value of the **LOCPATH** environment variable to locate an object containing a conversion method and tables. It will either use the method defined explicitly for the conversion or, if it cannot find one, it will use a default method with the tables defined explicitly for the conversion. Any conversion between two single-byte code sets can use the default method.

The **iconv**( ) function calls the conversion method in the object, which uses the tables to convert the data.

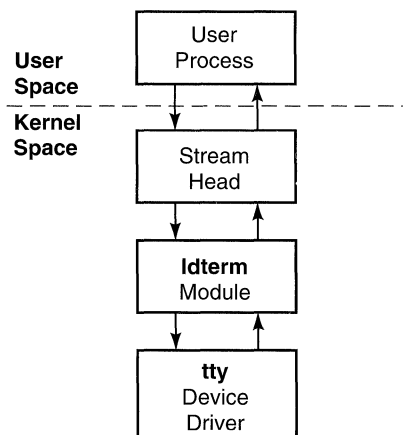
The **genxlt** command is used to create the conversion tables.

10.7 Terminal Device Support for Internationalization

One of the distinguishing characteristics of the internationalization subsystem is that it allows a user to select an application's code set at runtime. The OSF/1 terminal subsystem has been specifically designed to support this behavior.

The implementation of the terminal subsystem is STREAMS-based. STREAMS-based terminal devices are built of several software modules. When a terminal device file is opened, the STREAMS subsystem creates a bidirectional data path, or stream, for communications between the user process and the device. Figure 10-3 illustrates the stream for a terminal that is configured for operating in the C locale (the default locale provided with OSF/1).

Figure 10–3. Basic Stream for Terminal Devices



The stream in Figure 10-3 is composed of the following:

- The stream head module, which processes system calls made by user processes and controls the overall stream activities.
- The line discipline module, which interprets input and output to the terminal. OSF/1 provides **ldterm**, which is the standard line discipline for STREAMS-based terminal devices. This module uses the Extended UNIX Codes (EUC) encoding method for any data it processes. EUC defines a 7-bit ASCII character format and three multibyte character formats for applications and terminal devices to use. The **ldterm** module will accept data in any or all of these formats simultaneously. If the application or terminal device driver does not use an EUC code set and does not translate from this code set into EUC, another module must be added to the tty device stream for character conversion.

  Applications can use other line disciplines that are compatible with the OSF/1 tty subsystem.
- The hardware-specific device driver, which controls input and output to the terminal.

Other modules and software drivers can be present on the terminal device stream for any data processing and device control that is required.

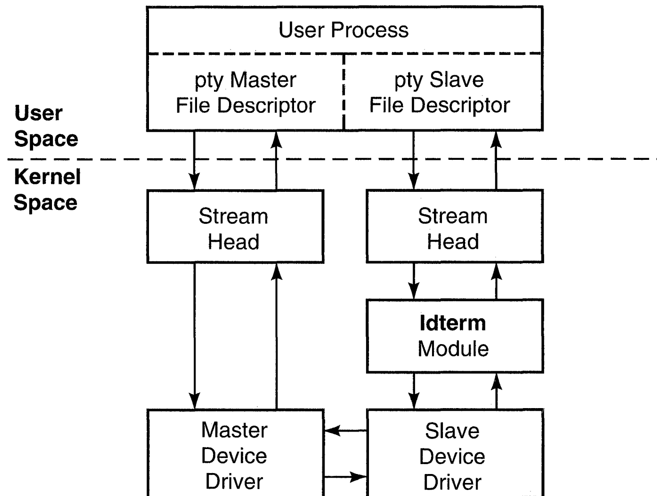
Some applications use pseudoterminals rather than real terminals. One example is X-windows. Another example is a process that remotely logs into another computer. Both applications operate in the client and server mode, where two processes communicate with each other without an intervening hardware device.

Just as it does for tty devices, the kernel automatically configures the stream for a pty when the device is opened. Applications that want to execute with a different device stream configuration must add and remove modules.

The pty subsystem presents the application with an interface identical to the tty subsystem. It defines two devices, called the "master" and "slave" devices. The slave device, which provides the interface to the user process, is manipulated by another process through the master half of the pty.

Figure 10-4 illustrates a basic pty device stream.

Figure 10–4. Basic Stream for Pseudoterminal Devices



To construct a pty device stream, the kernel actually creates two streams. First, it creates a stream for the master device with the driver **ptm*** at the bottom. Then, it creates a stream for the slave device with the driver **pts*** at the bottom. In addition, it pushes the **ldterm** module onto the slave device stream.

10.7.1 Initialization of Terminal Lines

The OSF/1 **autopush** facility allows for the automatic initialization of terminal lines. This facility consists of a system configuration file (the **autopush.conf** file as supplied by OSF/1) and the **autopush** command. The system administrator enters the names of the master and minor devices of the tty and pty devices defined in the system into the configuration file, along with lists of STREAMS modules to be pushed onto the device streams. Then the system administrator arranges for the **autopush** command to be run at system startup. When run, the **autopush** command loads the terminal configuration information into the kernel. Subsequently, any terminal device named in the kernel database will be automatically configured on device open.

10.7.2 Reconfiguring Terminal Lines

The terminal subsystem provides the **strchg** command to enable users to interactively configure their terminal lines to change which code sets are being used. For example, users could configure their terminals so that they handle SJIS character codes.

Code set converter modules are pushed on the stream in the order specified on the **strgch** command line. Converter modules can be placed between the terminal device driver and **ldterm** to convert characters back and forth between the keyboard and the line discipline module, and between **ldterm** and the stream head to convert characters back and forth between the line discipline module and the application.

Terminal streams can also be reconfigured within applications. An application uses the **I_POP ioctl**( ) call to pop **ldterm** and the other modules from the stream, and the **I_PUSH ioctl**( ) call to push the code set converter modules and **ldterm** on the stream.

Chapter 11

File Management

From the perspective of a user process, all objects that provide I/O are represented by files in the file system. Each file that a user process has access to is represented by a *file descriptor*. A file descriptor might represent an open file, a stream, a device, or a network socket, but a uniform set of file operations hides the distinctions between various files, devices, and networks.

This chapter describes the three components of the OSF/1 file management architecture:

Descriptor Management
: The data structures and functions involved in the management of open files.

Virtual File System
: A subsystem that provides a uniform means of access to the system's files, thereby allowing OSF/1 to support multiple file system types. The Virtual File System (VFS) translates generic requests on a file to the specific terms required by the file's file system. The VFS also supports the capability to dynamically add new file system types to the kernel.

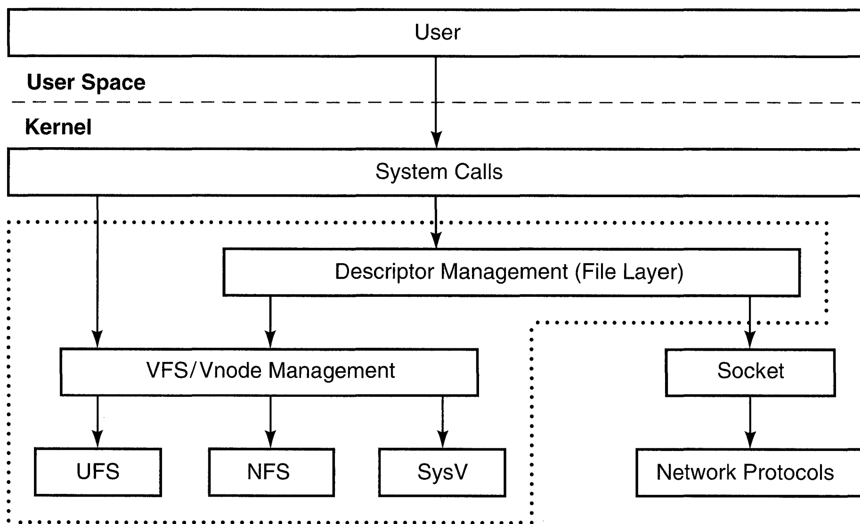
File Systems

OSF/1 provides three major file system types:

- UNIX File System (UFS)
- System V File System (SysV FS)
- Network File System (NFS)

Figure 11-1 shows how these components relate to one another. When a user process executes a system call to initiate an operation on a file, the system call causes the process to trap into the kernel. Here, the illusion of uniformity is preserved until the system call's request reaches the file layer, which distinguishes network sockets from file system objects. This chapter does not discuss sockets; they are discussed in Chapter 12.

Figure 11–1. Architecture of the File Management System

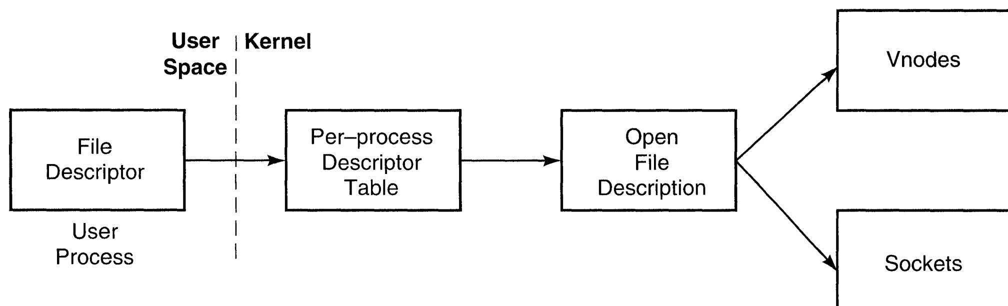


11.1 Descriptor Management

All user process I/O in OSF/1 is handled through descriptors. Every active socket or open file (including directories, links, special files, and so forth) in a user process is represented by a descriptor. The usual method for a process to acquire a new descriptor is through either the **open**( ) or the **socket**( ) system call. The process uses the descriptor to identify the socket or file when making an I/O request to the kernel. The file layer (see Figure 11-1) distinguishes between file descriptors that represent sockets and those that represent vnodes. It calls the appropriate I/O routines in each case.

Each descriptor specifies an entry in the *per-process open-file table*. That entry points to an *open file description* in the kernel that contains information such as the current offset, the descriptor type, the reference count, the set of available operations, and a pointer to the associated socket or vnode. Figure 11-2 shows that a file descriptor references a vnode (or a socket) by selecting an entry in the per-process description table, which in turn points to an open file description. The open file description specifies a vnode or a socket.

Figure 11–2. File Descriptor Reference to Open File Description



The per-process open file table is shared among all the threads of a process, and thus the threads share the same set of file descriptors. As soon as one thread obtains a new file descriptor, it is immediately seen by all the other threads in the process. When a process forks, its per-process open file table is copied to the child, thereby sharing all the open file descriptions. File descriptions subsequently obtained by either parent or child are not shared.

File descriptors can be marked **close-on-exec**, in which case they are closed if the process calls **exec( )**. Descriptors that are not marked **close-on-exec** remain open after a call to **exec( )**.

11.1.1 Data Structures

Figure 11-3 shows an example of open file descriptions. A process, P1, consists of three threads, T1, T2, and T3. These threads all share the same set of descriptors and the same per-process table. (The synchronization of thread access to the per-process data is discussed later in this chapter.) Among P1's descriptors are two that reference the open file descriptions D1 and D2. D1 points to a socket, and D2 points to a vnode.

Figure 11–3. A Process and Its Open File Descriptions

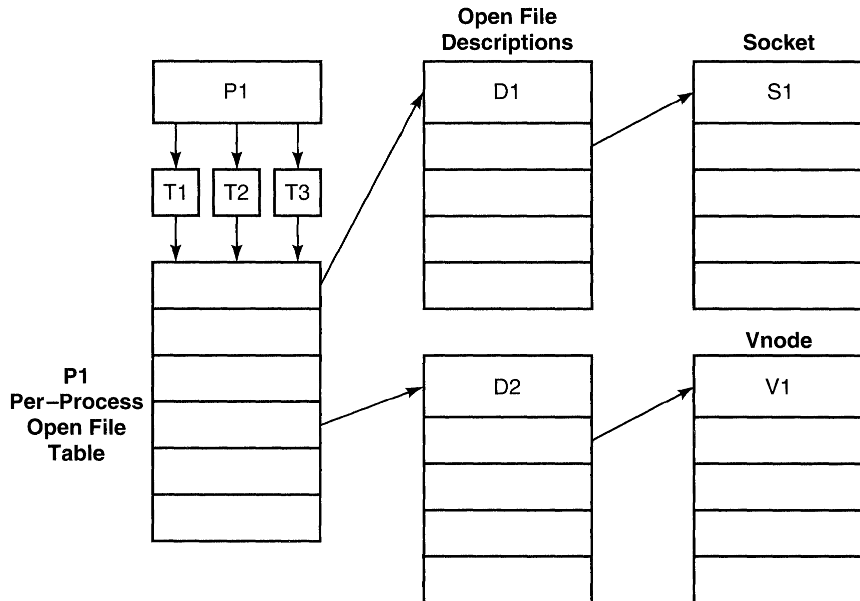
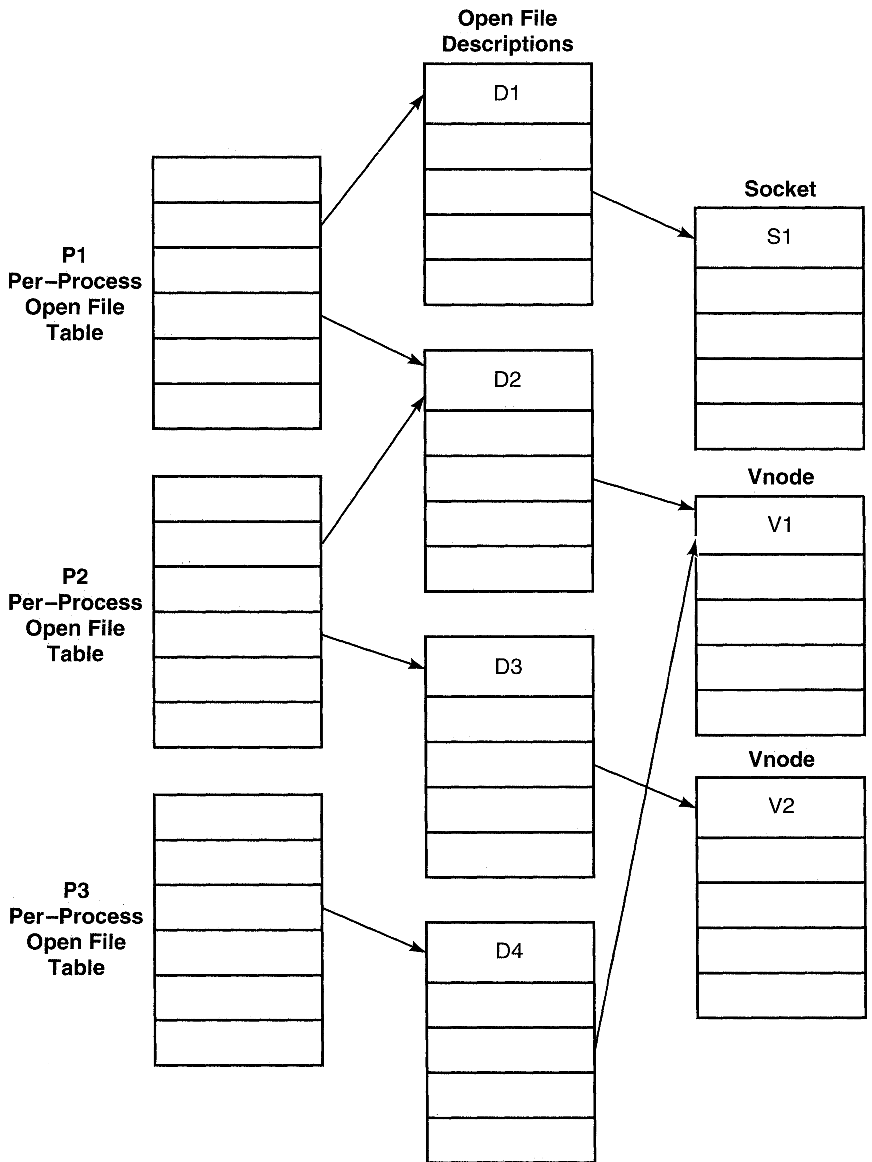


Figure 11-4 shows two additional levels of shared access. Two processes, P1 and P2, share access to the single open file description D2, and thus share the file object represented by vnode V1. The situation occurs when a process forks, leaving two processes holding the same description. Because they share an open file description, they see the same value for the current offset. Process P3 also shares access to V1, but through a separate open file description, having used an independent call to **open( )**.

Processes do not normally share sockets, except by inheritance through **fork( )**, because of the nature of sockets, which provide new, unique instances when created through the **socket( )** system call. For a discussion of sockets, see Chapter 12.

Figure 11–4. Processes Sharing a Vnode



11.1.2 Synchronization on Descriptors

All threads of a process share the per-process open file table. Threads using any of the following operations must synchronize on the table:

- Allocation and deallocation of file descriptors
- Access to the contents of the open file description itself
- Access to the file or socket

The second and third operations pertain to synchronization between all threads that share a particular open file description, not just those within a single process.

In the first case, allocation and deallocation of the file descriptors, a lock on the per-process open file table guarantees consistent information during the time the thread is looking for an empty slot in the table, or when it is retrieving an open file description through a valid file descriptor.

A special situation is worthy of note. If one thread were obtaining a file descriptor while another called **close()** to deallocate the open file description, the first thread could obtain a descriptor referring to nothing. To avoid this circumstance, the thread obtaining the descriptor increments the reference count on the open file description before releasing the lock, preventing deallocation. The reference is released when the operation is completed.

The second case, accessing the contents of the open file description, uses a lock to protect the description contents.

In the third case, file or socket access, all threads sharing access to an object represented by an open file description must be synchronized to preserve POSIX semantics. This synchronization protects the offset into the object, which is kept in the open file description. Any operation that affects the offset (such as **read()**, **write()**, and **lseek()**) takes a lock that is released when the operation has completed and the offset has been modified.

11.2 Virtual File System Management

This section covers the Virtual File System (VFS) layer and VFS operations, and is primarily concerned with entities at the file system level, as opposed to the individual file manipulation level.

The OSF/1 VFS provides support for multiple file systems of different types. File system types implemented under the OSF/1 VFS architecture include UFS, NFS, and System V. VFS support can be extended to include additional file system types, including file system types that are not based on the UNIX file system.

OSF/1 VFS implements the traditional UNIX file system interfaces for all file systems, regardless of their types. To a user process, these interfaces appear unchanged. A user process can use the traditional UNIX system calls to open, create, close, delete, and rename files and directories in any file system. To a user process, file systems of different types are indistinguishable.

The following terms are important to this discussion:

VFS architecture
: The mechanisms that enable using multiple file system types in OSF/1. The VFS architecture consists of two parts: file system entities and files through vnodes.

VFS layer
: The file-system-independent operations on file systems, such as **mount( )**, **sync( )**, and so forth.

VFS operations
: Higher level file system operations exported by file systems, such as **mount( )**, **unmount( )**, and **sync( )**.

Vnode layer
: File-system-independent operations on individual files, through vnodes, such as name translation, **open( )**, **read( )**, **write( )**, and so forth.

Vnode operations
: The vnode-level functions exported by file systems implemented under the VFS.

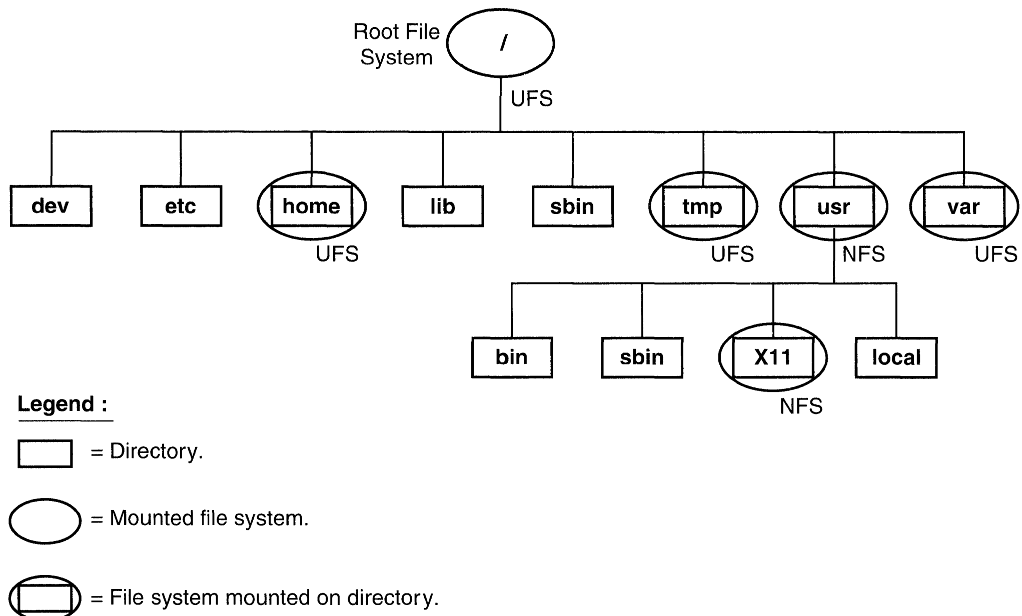
11.2.1 An External View of the File System Tree

The file system tree under VFS always starts with a root file system that is mounted when the system is booted. (The choice of the root file system is a VFS configuration issue.) Additional file systems come under VFS control when they are mounted onto the root file system or any other mounted file system.

Under VFS, mount points mark the boundaries between file systems. If a file system of one type is mounted on a file system of another type, VFS automatically switches its operations (both VFS and vnode) when a user program crosses the mount-point boundary.

Figure 11-5 shows an example of the basic OSF/1 file system tree.

Figure 11–5. Example of OSF/1 VFS File Tree



11.2.2 The VFS Switch

The virtual file system switch, or **vfssw**, is a data structure that represents all of the file system types currently available in an OSF/1 kernel. The **vfssw** is implemented as an array of VFS operation vectors, indexed by file system type. When performing a VFS operation, such as **mount( )**, the kernel uses the file system type (typically an argument to the system call) to find the correct VFS operation vector. Through this vector it calls the file system's VFS function. If no file system matching the *type* argument is present, an error is returned.

Although there are typically several file systems statically loaded into the **vfssw**, OSF/1 allows file systems to be dynamically loaded and unloaded. This implies that the **vfssw** can change at run time. As a result, there are synchronization requirements placed upon **vfssw** access. Modifications to the **vfssw** must not take place during a mount operation; otherwise, it would be possible to mount a file system that is being unloaded, or to fail a mount of a file system that is being loaded. To prevent such errors, a lock protects the **vfssw**. Operations that change the **vfssw**, such as adding and deleting file system types, hold this lock for writing, while the **mount( )** operation holds it for reading, preventing modifications while mounting is in progress.

11.2.3 Internal Representation of Mounted File Systems

In OSF/1 there may be any number of mounted file systems, up to a system-defined maximum. Each file system type is responsible for setting its own limit on the number of mounts allowed.

Each mounted file system is represented by a **mount** structure. The **mount** structure is divided into two major sections—file-system-independent and file-system-dependent. The **mount( )** operation allocates and initializes the file-system-independent portion, and the **umount( )** operation, if successful, deallocates it. Individual file systems must allocate, initialize, and deallocate their own file-system-dependent information. Mount structures are maintained on a list, one per mount instance.

In addition to the list, the file-system-independent section contains the following significant fields:

Operations vector
: A pointer to the VFS operations vector for the file system

Covered vnode
: A pointer to the vnode of the file which serves as the mount point for the mount instance

Node list
: A list of vnodes associated with the file system

Status
: A **statfs** structure describing both static and dynamic status information related to the file system

Data
: A pointer to file-system-specific mount information; this is not accessed by the VFS

Locks
: Several locks for synchronizing access to the **mount** structure and to the file system it represents

The **mount** structure is not exported to user level, and should never be examined by user programs. A user who requires information on mounted file systems can use any of several system calls that access the mount information, such as **statfs**( ), **fstatfs**( ), **getfsstat**( ). These calls pass the information in the exported structure, **struct statfs**.

OSF/1 places several semantic restrictions on the mount and unmount operations:

- A single physical file system cannot be mounted in more than one place. This restriction does not apply to some remote file systems. For example, it is possible to mount the same remote file system using NFS in more than one place in the local file system tree. Because this can cause problems with respect to buffer cache consistency, it is not a recommended practice.
- Generally, it is not possible to unmount a busy file system. If any vnodes on a file system are active (that is, they have a reference count greater than 0 (zero)), the unmount operation will fail. Forcible unmount of file systems is a feature of the VFS architecture; however, part of its implementation is file-system-dependent, and no file system supplied with OSF/1 supports forcible unmount.

- A block device, that is, a physical file system, that is currently open cannot be mounted. Conversely, it is also not possible to open a mounted block device. These restrictions exist to maintain buffer cache consistency.

11.2.4 Pathname Translation from Name to Vnode

As mentioned earlier, all open file descriptions reference either a socket or a vnode. All I/O activity in OSF/1, with the exception of sockets, passes through the vnode layer. For every regular file, directory, or device file that is active, there is exactly one vnode representing it.

Operations on files can be grouped into two categories:

- Operations that access files by filename
- Operations that access files by descriptor

Named file operations typically result in a translation of the filename to a vnode. This name translation is a central function of the VFS implementation. Once the name has been translated, the file may remain open (as by the **open( )** system call), in which case it remains referenced by an open file description, or it may simply be examined for status (as by the **stat( )** system call), and its reference released. Examples of operations that use file descriptors are **read( )** and **write( )**. These operations take open file descriptions that reference previously translated vnodes as arguments, and operate on file data.

The name translation mechanism takes a pathname as input and returns a referenced vnode. In OSF/1, this mechanism works at two levels: in the **namei( )** function of the the VFS layer, and as a vnode operation supplied by individual file systems. The following discussion primarily concerns the VFS level operation of **namei( )**.

11.2.4.1 The namei( ) Function

The **namei( )** function performs translation of a pathname to a vnode. It is the central name translation routine in OSF/1, and can be summarized as follows:

1. Optionally copy the pathname to an internal buffer. (A flag specifies whether this has already been done.)
2. Get a starting directory for the lookup routine; this is typically either / (root) or the process's current directory.
3. Loop doing the following:
   - Copy the next component to a buffer.
   - If the path is **..** (dot, dot) and the lookup routine is at the root of a mounted file system, find the parent vnode to cross the mount point.
   - Call the (file-system-specific) lookup routine for the next component. This function returns a referenced vnode.
   - If the vnode represents a symbolic link, copy the name to the internal buffer and continue to loop.
   - If the vnode is a mount point for another file system, find the root vnode of the mounted file system (using the **VFS_ROOT** VFS operation) and continue to loop.
   - If there are more pathname components, loop.
4. Return the referenced vnode.

Of course, an error may occur at any stage, in which case **namei( )** returns the error.

Its activity may be summarized as, "look for pathname component *xxx* in the directory represented by vnode *dvp*," iterated through all the components of the pathname. For example, the translation of the name **/usr/bin/ls** would proceed as follows:

1. Set *dvp* to the vnode for /, which is well-known.
2. Look up **usr** in *dvp*, setting *dvp* to the vnode for "/usr."
3. Look up **bin** in *dvp*, setting *dvp* to the vnode for "/usr/bin."

4. Look up **ls** in *dvp*, setting *dvp* to the vnode for "/usr/bin/ls."
5. Return this vnode.

In the simple case, in which there are no mount points and no symbolic links, **namei**( ) has little extra work to do. In the other cases, it must perform some logic, summarized as follows:

Mount points
: A vnode that identifies a mount point has a special field that points to the covering mount structure. When such a vnode is returned by the lookup function (which is file-system-specific), it must be translated into the root vnode of the mounted file system. This extra translation is the reponsibility of the VFS operation, **VFS_ROOT**, provided by the file system.

Symbolic links
: A symbolic link is a particular type of file that contains a pathname as its data. When **namei**( ) encounters a symbolic link, this pathname is the one to be translated. It copies the new name into the internal buffer and continues translation.

Parent of mount
: When the current directory in **namei**( ) is the root of a mounted file system (other than /), and the current component is the parent directory, (**..**), then **namei**( ) must traverse the mount point in the reverse direction. To do so, it must find the vnode of the covered directory for the mount point, and perform the translation of **..** starting there.

In OSF/1, the input for both **namei**( ) and the file-system-specific lookup operations is the entire pathname being translated. The file system determines which portion of the pathname it can correctly translate at one time. Typically, a file system translates one pathname component at a time; however, it may choose to do more. For example, if a local file system does the work to correctly recognize and handle symbolic links and mounted file systems, then it could translate an entire pathname in one call to its lookup routine.

There are other instances in which a file system may have specific knowledge of the contents of a pathname that allow it to efficiently translate multiple components in one call. An example of this is a distributed file

system that has a well-defined name space and requires expensive name server calls to translate the components of pathnames. By sending the entire pathname to the name server in a single call to translate multiple components, it could gain significantly in performance, especially if the local name cache is not used.

11.2.4.2 Pathname Translation and Mount Synchronization

Mount and unmount operations change the name space visible during pathname translation. As a result, it is important that the view of the name space remain consistent while mount, unmount, and pathname translation operations are taking place. OSF/1 maintains this consistency in several ways.

First, newly created mount points have no effect on pathname translation until they are fully initialized. This ensures that a translation in progress will either use the old file or the new mount point, and not something in between. In addition, the pathname translation does not need to wait for the mount operation to complete.

Also, an unmount operation must first make sure that there is no activity in the file system before dismantling the mount point. Once an unmount operation has determined that the file system is inactive, the file system must remain inactive until the unmount is complete. The types of activity that must be synchronized with unmount include normal pathname translation, file system synchronization (which traverses the list of mounted file systems, writing dirty buffers to stable storage), and the VFS operation, **VFS_FHTOVP**, which translates a file handle to a vnode. The **VFS_FHTOVP** operation is logically equivalent to pathname translation, using a file handle instead of a pathname.

All of the operations that are sensitive to unmount operations must hold a lock during critical sections of their code. This lock prevents unmount from causing inconsistencies. Once the unmount operation obtains the lock, operations such as pathname translation, which attempt to access the affected mount point, are blocked until the unmount has completed, either successfully or not.

Note that typically, if a file system has active (referenced,) vnodes, it cannot be unmounted. However, certain file system types in OSF/1 can be

unmounted, even if they have active vnodes (for example, the file-on-file system). The VFS level, as well as the file systems themselves, must cooperate and be careful to make sure that no inconsistencies arise from this type of unmount.

11.2.4.3 The Name Cache

Pathname translation is a frequent activity. Because it involves successively reading directories and inodes from disk, it is quite expensive. Access through remote file systems, over a network, can be even more expensive. OSF/1 minimizes the work of pathname translation by caching the names found by directory scans for future reference.

OSF/1's name cache is available to all file systems. The cache is indexed by a hash value on the pair *vp*, *name*, where *vp* is the vnode that refers to the directory containing *name*. The cache management algorithm removes the least recently used names to make room for new names, ensuring that frequently used names remain available. OSF/1 also makes a cache entry when a pathname cannot be translated to a vnode because the pathname does not correspond to an existing file. This is called *negative caching*.

Name cache references to vnodes do not increment the vnodes' usage counters, and do not prevent the vnodes from being recycled to a different, underlying file system type.

When a vnode is recycled, it is not economical to search the name cache for obsolete entries in order to invalidate them, because several different name cache entries may refer to a single vnode. Instead, the vnode gets a new, unique **v_id** field, so that future searches through the cache to the previous incarnation of the vnode will fail. As a result, memory allocated to a vnode cannot be deallocated unless a separate action is taken to ensure that all cache references to the vnode have been removed.

11.3 Vnode Management

The vnode is the fundamental data structure of the VFS architecture.

Each active object (directories, devices, regular files, symbolic links, FIFO special files, and so on) in an OSF/1 file system is represented by a unique vnode structure. The vnode structure is divided into two sections:

File-system-independent
: Contains fields that are required by all objects, and that are used primarily by the VFS and vnode layers of the VFS architecture.

File-system-dependent
: Contains information specific to a file system. This section is maintained by file system implementations.

11.3.1 The Contents of a Vnode

Most vnodes are allocated and activated during pathname translation. Before a file system first activates an object, it must allocate a vnode with which to associate the file-system-dependent information. OSF/1 maintains a list of vnodes that are available to all file system types. Once a vnode is initialized to a specific file system type and is referenced, it is no longer available to other file systems. In addition, its type may not change, except under the specific conditions of forcible unmount and character device revocation, which are discussed in other sections of this document.

When a vnode is no longer referenced, it is placed on a free list, making it available to other file systems. However, its contents remain intact, and the file system may reactivate it by referencing the same file again, thus removing the vnode from the free list. Vnodes that remain on the free list are eventually recycled to other uses. This mechanism allows file systems to cache information for frequently accessed files.

The following list describes some of the important file-system-independent fields of the vnode structure.

Flags
: Contains state information, such as whether the object is a root directory, the file locking state, and vnode transition states.

Reference counts
Each vnode contains two reference counts:

- Active references (for instance, open file references)
- Buffer cache buffers that reference the vnode

The first type of reference is considered "hard" and may not be released by the vnode architecture. It is up to the user of the vnode to release such references, for example, upon closing a file. The second type of reference is considered "soft." These references are primarily informative and are released when the buffer is flushed.

File locking information
A count of shared and exclusive file locks for use by flock-style file locking. POSIX style advisory file locking and SVID-compatible mandatory file locking are handled separately, and are indicated by vnode flags fields.

Capability identifier
An identifier field that is reset when a vnode is recycled. A cache hit is not valid unless the cache entry's capability identifier matches that of the vnode.

Type field
Vnodes have types. The type is established upon initialization and cannot change during the lifetime of the vnode, except in the cases of forcible unmount and character device revocation. The following types are used by the OSF/1 VFS:

VNON An allocated but still untyped vnode

VREG A vnode representing a regular file

VDIR A directory vnode

VBLK A block device vnode

VCHR A character device vnode

VLNK A symbolic link vnode

VFIFO A FIFO special file vnode

VSOCK A vnode representing a UNIX domain socket

VBAD An invalid vnode

Vnodes of any type, including **VBAD**, may be referenced or free. The vnode operations vector of a vnode of type **VBAD** contains functions that return errors.

Operations vector
: A pointer to a vector of vnode operations. This vector is specific to the file system and vnode type of the vnode.

Mount structure pointer
: A pointer to a mount structure. A valid vnode typically points to the mount structure of the file system containing the object that it represents. Invalid vnodes (of type **VBAD**) always point to a default mount structure, called the **DEADMOUNT**. The **DEADMOUNT** structure is used as a placeholder, and has VFS operations that return errors. The mount pointer in the vnode is set up when the vnode is allocated and initialized by a file system type.

Various lists
: Lists that contain the vnode as an element. Every vnode is on a number of lists at any given time, including the following:

  Vnode free list
  : If a vnode's reference count is zero, it is on the vnode free list; if its count is non-zero, it is not.

  Mount vnode list
  : If a vnode is active on a specific file system, it is also on the list of active vnodes for that file system.

Offset of last read
: The current byte index into a file being read.

Buffer cache
: Lists of buffers that reference the vnode of the object for which they contain data. The vnode also references the buffers. There are two lists of buffers for each vnode: clean and dirty. These lists are traversed by operations such as **sync( )**, **fsync( )**, and **umount( )**.

Reader and writer counts
: Some operations and FIFO special files require reader and writer counts for an object. These counts are kept in the vnode. They are incremented on opening and decremented on closing, as appropriate.

Virtual memory private information
: A vnode may be mapped, by either **mmap()** or **exec()**, or it may represent a paging file. If a vnode interacts with the VM system, it contains a valid pointer to a VM data structure.

Union of pointers
: A union of pointers that are conditional upon type and state of the vnode. If the vnode is a special file, the union is a pointer to a **struct specinfo**. If the vnode represents a UNIX domain socket, the union points to a **struct socket**. If the vnode has a file system mounted over it, then the union is a pointer to the mount structure of the covering file system.

File system private data
: Space available to file systems for their private use. While this space is currently allocated in the vnode structure, file system implementations should make no assumptions that the data is contiguous, and should properly abstract the translation between the vnode and the file system node (for example, the UFS inode).

Various locks
: The vnode contains locks to protect its own contents and lists.

11.3.2 The Free List and Cache

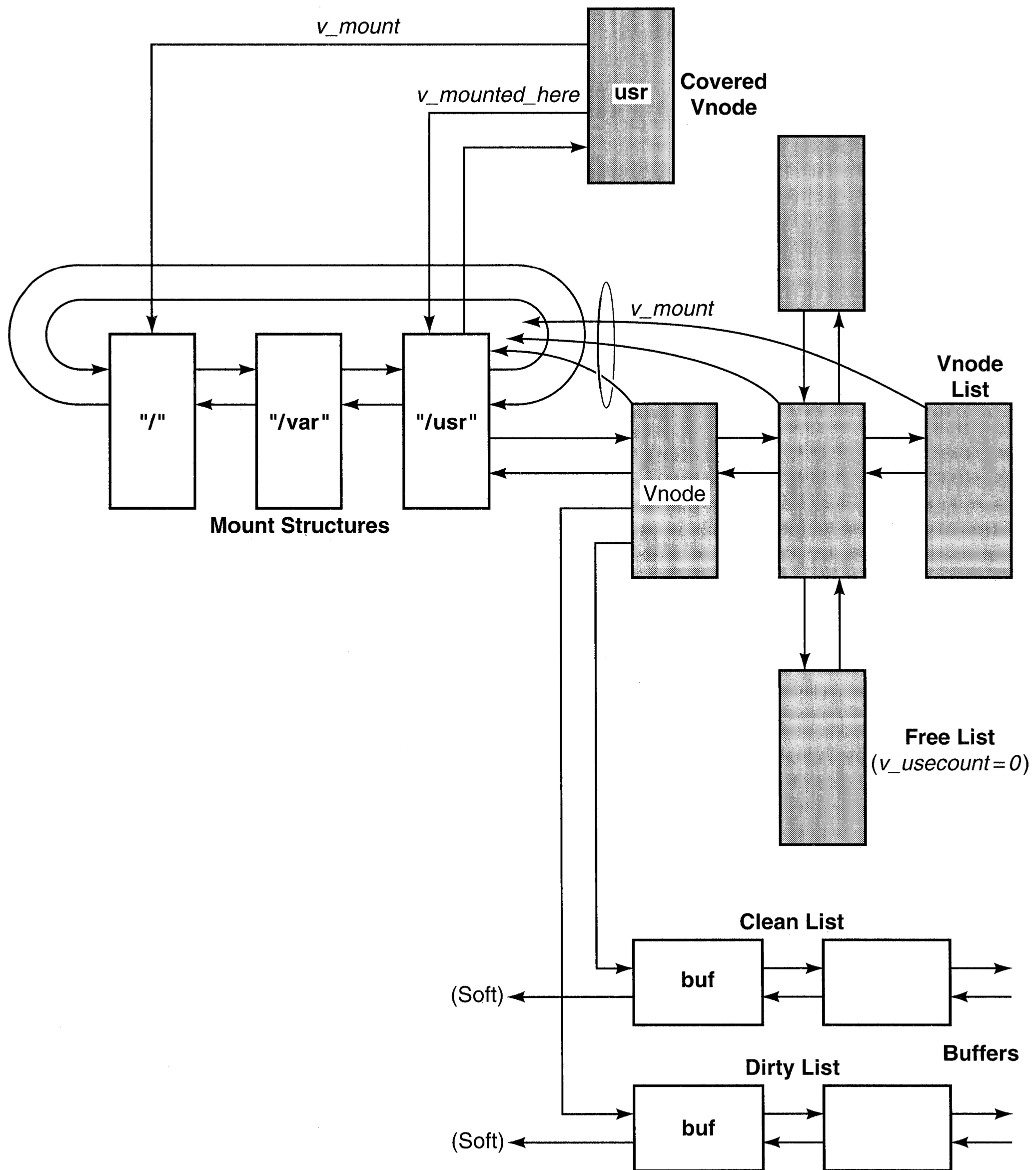
The vnode architecture allows inactive vnodes (those having a reference count of 0 (zero)) to be valid, that is, initialized. It also allows them to contain cached information, in the form of file system private data and buffer cache buffers. Because recreating a vnode can be an expensive operation, it is preferable not to destroy old vnodes if they might be reused. This caching mechanism is implemented as part of vnode management in OSF/1.

The vnode free list is more than simply a list of available vnodes. It is managed as a least recently used (LRU) list of inactive vnodes. When a new vnode is required, it is removed from the front of the free list by the function **getnewvnode( )**. When a vnode's reference count goes to 0 (zero), as detected in the function **vrele( )**, it is placed on the end of the free list. This way, a vnode will remain on the free list for some period of time before being recycled. Recycling destroys any cached information present, and breaks the vnode's association with a file system.

11.3.3 The Life Cycle of a Vnode

The following discussion outlines the life cycle of a typical vnode in OSF/1, starting with its allocation, and ending with its recycling to a new type. Figure 11-6 illustrates some of the structures involved.

Figure 11–6. Example of Data Structures for a Mounted File System



Vnodes are typically allocated when a file system calls the function **getnewvnode( )** during pathname translation. Upon allocation the vnode is removed from the free list and appropriate fields are initialized. The reference count is set to 1 (one). The appropriate operations vector is passed as a parameter to **getnewvnode( )**. At this time, the vnode appears on no lists; although it may be referenced by the name cache, the cache reference is invalid because the capability identifiers no longer match.

At this point, the file system implementation initializes its private data in the vnode, typically by reading an on-disk or remote data structure, such as an inode or nfsnode. The file system is also responsible for initializing several other fields of the vnode. It must initialize the type of the vnode, and add the vnode to the list of vnodes valid on a mounted file system. If the file being accessed is a device special file, then the file system must call a function to initialize device-specific information. If the file is of a type other than **VREG**, **VDIR**, or **VLNK**, the operations vector may also need to be changed.

Most file systems cache their nodes (inodes, for instance) for quick access. This is also done at vnode initialization time. If the initialization is on behalf of pathname translation, then the referenced vnode is returned. At this time, the vnode may be on the valid vnode list for the mount point, the list of valid nodes for the file system type, or both.

The vnode is then available for accessing the object it represents. If the file is open for reading and/or writing, the vnode is of the appropriate type, and the file system implementation takes advantage of the buffer cache, then the data read or written may be cached, adding buffers to the lists of clean and dirty buffers associated with the vnode. Dirty buffers are flushed and put on the clean list during **sync( )** operations.

When the final reference on a vnode is released, for example, by a **close( )** operation, the vnode is placed on the vnode free list. Before the vnode is put on the free list, the file system implementation is given an opportunity to perform its own inactivation through its **vn_inactive( )** function. A typical **vn_inactive( )** function performs functions such as deallocating resources associated with the file if it has been removed.

Once the vnode is on the free list, it is available for recycling, but it may still be reactivated from the free list any time before it is recycled. For example, if its name is translated again, the file system may get a cache hit and call the function **vget( )**, which simply reactivates the vnode, removing it from the free list. When a vnode has been recycled, there are no

references to it other than a potential soft reference from the name cache, and it may safely be reallocated to another file system or type.

Any of several operations may cause invalidation of the vnode. Among these are forcible unmount, character device revocation, and vnode recycling. When a vnode is invalidated, as during recycling, it must first be cleaned. Cleaning involves removing it from any lists it may be on and flushing any cached information. Because some of the information may be file-system-specific, the file system's **vn_reclaim()** operation is called. A typical **vn_reclaim()** operation removes a vnode from its hash chains, and disassociates the file-system-dependent data from the vnode.

11.3.4 File Locking

OSF/1 supports two different styles of file locking—POSIX file and record locking, and the Berkeley flock-style file locking. Because some file systems may require some interaction when a file is locked, the POSIX file locking is implemented below the vnode layer, adding a vnode operation to the vector. The vnode layer provides file-system-independent functions that may be called to lock and unlock files.

11.3.5 Special Files

The OSF/1 VFS architecture provides file-system-independent operations for FIFO and device special files. A file system implementation may choose to support or not support special files. A file system supports special files by calling vnode operations specific to special files. These operations are exported by the vnode layer. If a file system supports special files, some intervention on the part of the specific file system may be necessary. An example is updating the modification times on inodes. POSIX specifies that file modification times must be updated on writes. As a result, the file system may need to intercept the **vn_write()** operation, mark its node for update, and then call the file-system-independent operation. A file system must attach the appropriate vnode operations vector at the time the vnode for a special file type is initialized.

11.3.5.1 Device Special Files

Block and character special files in OSF/1 are managed by a set of device-specific operations. In OSF/1, as in most UNIX systems, there may be more than one device file referencing the same physical device. Device semantics require that all references to a specific device be associated, even if they originated with different files, and therefore different vnodes. This is true for both block and character special files.

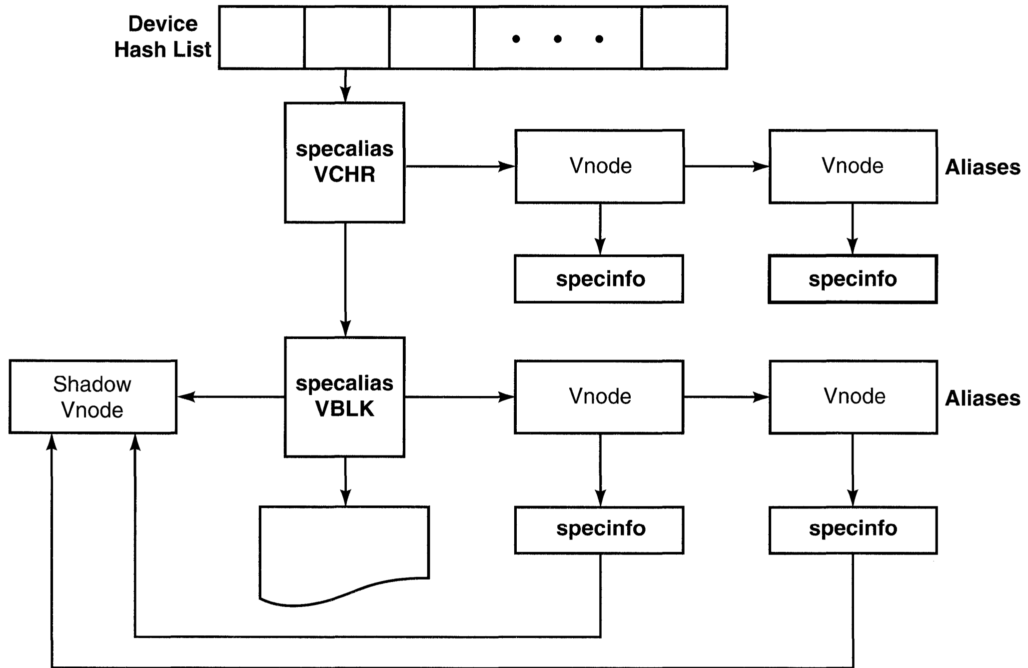
When a vnode is initialized for a device special file, the file system initialization operation must call the function **specalloc()**, which allocates and initializes a device-specific data structure, **specinfo**, that is attached to the vnode. It is at this time that the file system attaches the vnode operation vector for dealing with special files. When the vnode is subsequently opened by a call to the special file open operation, it is then associated with all other vnodes that reference the same physical device. The members of such a set of associated vnodes are called *aliases*. If a file system chooses to bypass the special-file open operation provided, it must perform its own association.

Character devices must be associated because it is necessary at times to forcibly revoke access to a character device, such as a tty. This is done by invalidating all vnodes associated with that device, causing further references to return errors. Subsequent translations of the same name result in the allocation of new, valid vnodes.

Vnodes that reference the same block device are associated for buffer cache consistency. Read and write operations on block devices use the buffer cache, and buffers are cached based on pairs of **vnode**, **offset**. If more than one vnode (that is, more than one file) were used to access the same block device, the same physical block could exist in two different cache entries, causing inconsistency.

Mounted local file systems use a block device for accessing file system metadata, such as inodes. For this reason OSF/1 does not allow the block device of a mounted file system to be opened, or a file system that has an open block device to be mounted. Buffer cache consistency between multiple instances of a block device is maintained by associating all related vnodes with a shadow vnode, which is used for all buffer cache activity. As a result, all buffers are associated with this shadow vnode, rather than any of the actual vnodes representing the device files. Figure 11-7 shows these data structures as used for special files.

Figure 11–7. Device Special Files Data Structure



Devices are hashed by device number (**dev_t**) into a list of pointers to **specalias** structures. There is exactly one **specalias** structure allocated per open device. The **specalias** structure contains a list of all vnodes that represent its device. Since the same **dev_t** may represent both a block and a character device, the **specalias** structure is typed either as block (**VBLK**) or as character (**VCHR**).

11.3.5.2 Clone Devices

OSF/1 supports the notion of a cloned character device. A clone is a new instance of a character device, which is not necessarily associated with a particular file in the file system tree. Any character device driver may be selected to be clonable. When clonable devices are opened, they cause the allocation of a new, unique device number (**dev_t**), which is not associated with that of the original file. This also results in a new vnode, which is also unrelated to the vnode representing the file used for the initial open call.

The cloning mechanism is useful for drivers that need to create a new instance of a device on each open, and to avoid the necessity of creating a multitude of unique device special files in a file system.

Cloned vnodes are not associated with any specific file system. Because they represent devices and not files, they have no need for the association. However, certain operations are required to take place, such as the updating of modification times and the correct operation of system calls such as **fstat( )**. As a result, cloned vnodes have their own vnode operations vector, which implements the required semantics.

11.3.6 The Buffer Cache

The buffer cache increases file system performance by storing copies of file system data in memory. The relationship between the vnode and buffer cache is illustrated in Figure 11-6 and Figure 11-8.

The OSF/1 buffer cache implementation differs from older UNIX systems in that the OSF/1 buffer cache is accessed by *vnode, logical block number* pairs, rather than by *device, physical block number*. Buffers are tagged with the vnode representing the object whose data they contain. Buffers containing data that is not associated with any particular object (for instance, file system data structures such as inode data, or an indirect block) are tagged with the vnode of the block device of the file system. This is not applicable to file systems with no local storage, such as NFS.

11.3.6.1 The Buffer Header

The buffer cache is organized as a series of hash chains that point to lists of buffers. All buffers not currently in use are linked together on a free list. Buffers also reside on clean and dirty lists headed by the vnode representing the file that contains the data in the buffer. The buffer cache hash chain header includes:

- Flags matching those of the buffer structure
- Forward and backward chain pointers
- A lock and timestamp for multiprocessor synchronization

The contents of each buffer and the data to which it refers are guarded by a mutual exclusion lock. The buffer lock preserves the semantics of the original uniprocessor buffer cache implementation by permitting only one thread at a time to use the buffer.

The lock and timestamp contained in the buffer cache hash chain header are used for synchronizing access to the hash chain, and for dealing with simultaneous attempts to allocate a buffer representing the same file data block.

11.3.6.2 The Buffer Free List

The buffer free list is an array of buffer header structures used as headers for the free list chains. There are several elements in the free list array:

- An LRU chain of useful buffers (as proven by their access patterns)
- A list of buffers that have not yet proved useful (aged)
- A list of empty buffers
- A list of locked buffers

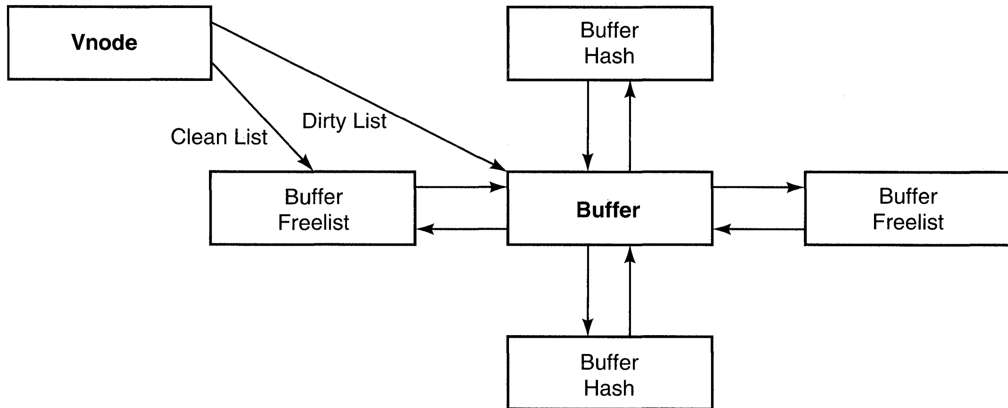
The LRU list is a strict LRU chain of buffers containing valid data. The second, or aged, list contains buffers that contain valid data, but have not yet been accessed, and therefore have yet to prove their usefulness. This list contains buffers, for example, with read-ahead data. The empty list contains buffers that have been returned to the pool, but have no associated memory. OSF/1 does not currently use the locked list.

11.3.6.3 The Buffer Structure

The buffer structure itself includes

- Flags.
- Hash chain pointers.
- Free list pointers.
- Vnode buffer list pointers.
- The block transfer count.
- The buffer size.
- A union containing the address of the pages containing the buffer data. This may be strictly file data, or it could be file system data, such as a superblock, cylinder group, inode list, or indirect block.
- The logical block number.
- The physical block number.
- The buffer's dirty region.
- A pointer to the vnode representing the file or file system with which the data is associated, along with references to read and write credentials.

Figure 11–8. Buffer Cache and Vnode Data Structure Interaction



11.4 The File System Layer

This section describes each of the several file systems that are provided with OSF/1, the derivation of each, and the changes made for operation in the OSF/1 VFS architecture.

11.4.1 NFS

The NFS implementation in OSF/1 is derived from the implementation of NFS in the 4.3BSD Reno release, which is derived from code contributed by the University of Guelph, Ontario, Canada. The NFS implementation is based on NFS protocol version 2. It does not include the extended mount protocol. It also does not contain any code to deal with file locking, such as a lock daemon. The major changes in the OSF/1 implementation are for parallelization of the code to enable it to operate efficiently on multiprocessor systems. Additional changes are for conforming to the OSF/1 VFS architecture.

11.4.1.1 Client Parallelization

NFS is a stateless protocol, making it relatively simple to parallelize. The implementation makes few assumptions about state, which can complicate parallel operation. The buffer cache synchronization effectively parallelizes all file data access. Other information stored by the NFS client, which may be shared by multiple threads, includes cached attributes and network connection state. The parallelization of the networking framework in OSF/1 provides most of the synchronization required by the network transport portion of NFS. OSF/1 provides mechanisms that synchronize access to other shared information in the NFS client, such as hash chains of NFS nodes and NFS nodes themselves.

Although the buffer cache synchronization is sufficient to provide the semantics guaranteed by the NFS protocol, most UNIX applications are accustomed to reading and writing operations that work atomically. This means that multiple read and/or write operations initiated on the same section of a file should not be interleaved below the level of the system call interface. This guarantee cannot be made in NFS, since multiple client systems may be accessing the same file, and large read and write operations may be fragmented when the NFS is involved. OSF/1 attempts to provide some atomicity by guaranteeing read and write synchronization between requests initiated from the same client. This is achieved with a lock on the NFS node representing a remote file in the client.

11.4.1.2 Server Parallelization

Parallelization of the NFS server is limited primarily to synchronizing the access to data structures shared between multiple server processes. The networking parallelization provides synchronization between requests received by the NFS server processes. It also provides the synchronization of sending reply messages. Once a request is received by a server process, it services the request in a way that interacts scarcely with other servers. Servicing a request involves calls to the local file system through the VFS and vnode interfaces, which provide their own parallelization guarantees.

11.4.1.3 VFS and Vnode Interface Changes

The absence of a vnode-level lock in OSF/1 has little effect on the NFS implementation, and removing the lock operations resulted in little code change. The lock on the NFS node suffices as a substitute.

11.4.1.4 Dynamic Configuration Changes

In OSF/1, NFS is dynamically configurable. The configuration manager daemon calls the **nfs_configure( )** routine, which calls **vfssw_add( )** to register the NFS VFS operations vector in the virtual file system switch (**vfssw**). NFS also dynamically adds two system calls to OSF/1 by calling **syscall_add( )**. The added calls are **async_daemon( )** and **nfssvc( )**.

Dynamic unconfiguration removes the operations vector from the **vfssw** by calling **vfssw_del( )**.

11.4.2 UFS

The UFS implementation in OSF/1 is derived from the 4.3BSD Reno release. The physical file system is that of 4.3BSD Tahoe, which is changed from the traditional 4.2BSD file system, but is compatible; this means that the OSF/1 UFS understands both the 4.2 and 4.3 Tahoe physical file systems. The major differences between the Reno implementation of UFS and the OSF/1 UFS relate to parallelization and changes to the vnode interface itself, for example, elimination of the lock and unlock vnode operations.

This section describes the implementation of UFS in OSF/1 in terms of differences from its origin—the Berkeley implementation. It is not intended to be a tutorial on UFS.

11.4.2.1 Parallelization

UFS in OSF/1 is fully parallel. This means that multiple threads can correctly execute UFS code simultaneously. Parallelization of UFS involves synchronizing access to all shared data and data structures in UFS. The data structures involved are

- Superblock
- Cylinder groups
- Mount structures
- Inodes (in-core, and as on-disk structures)
- Inode hash chains
- File and directory data
- Quota data structures

Synchronization of the superblock is achieved in two ways. First, the superblock is read from the block device into a locked buffer cache buffer. It is then copied to an in-memory data structure, which is guarded by a lock. Virtually all references to the superblock (other than updates) use the in-memory version.

In UFS, cylinder group information is file system metadata accessed by reading blocks from the block device representing the mounted file system. Because access to these blocks is through the buffer cache, the buffer cache synchronization is sufficient to provide mutually exclusive access to these data structures.

UFS-specific mount structures are accessed under the control of a lock protecting their contents.

Inodes have two parts, the on-disk data structure, or *dinode*, and the in-memory structure, or *inode*. The inode in memory contains a copy of its associated dinode. When dinodes are read into memory, they are accessed through the buffer cache, and thereby locked. They are then copied into the in-memory inode, whose content is guarded by a lock. All future access to the inode is protected by this simple lock. Updates to on-disk inodes are protected by both this lock and the buffer cache synchronization. UFS inodes are kept on hash chains for both caching and fast lookup. The hash chain data structures are protected with simple locks, one per hash bucket, which are taken when the chains are being examined.

Access to file data associated with an inode is synchronized with a lock contained in the inode. This lock is taken for reading when the data is being read. The lock is taken for writing when the data is being modified, for example, by a write or truncate operation. This lock also protects the size field of the inode. Although the synchronization on the buffer cache data is sufficient for file data, atomicity of reads and writes can be guaranteed only if the inode is locked while its data is being modified.

The UFS quota implementation in OSF/1 is also parallelized. There are two locks protecting quota data structures.

11.4.2.2 VFS and Vnode Interface Changes

Because there is no vnode level lock, UFS must perform its own locking in cases where the Reno code assumes the vnode is locked. The inode lock is sufficient for this use. The greatest effect of the absence of vnode lock and unlock operations is in file creation, deletion, and renaming. In the previous model, directory information was locked for extended periods of time, allowing the file system implementation to make assumptions about the state of the directory across multiple vnode operations, such as **lookup( )** and **create( )**. In OSF/1, no such guarantees are possible. Instead, the file system must detect changes in state and act accordingly.

An example is file creation. When performing pathname translation, the UFS lookup operation caches information about holes in directories where new directory entries may be created. This information is then used by the create operation to write the new entry. The caching allows UFS to skip an extra, expensive directory scan operation. In 4.3BSD Reno, the directory vnode is locked between the lookup and the create, guaranteeing the consistency of the cached information. In OSF/1, the directory is not locked. In order to avoid the extra lookup operation, UFS must detect changes in the directory between the lookup and the create operations, by examining a timestamp in the directory's vnode. If it has changed, then the expense of the extra lookup must be incurred. If it has not changed, then the cached information is valid and may be used.

11.4.2.3 Fast Symbolic Links

A symbolic link in BSD systems is created by writing a pathname into the data blocks of a file and marking the file type as VLNK. This type is interpreted by the pathname translation mechanism, and the data is read into the translation buffer as part of the pathname. Reading a traditional symbolic link may require two disk operations, one to read the inode of the symbolic link and one to read the file data.

OSF/1 UFS contains an optimization for symbolic links that can reduce this I/O burden. If the data for a symbolic link is less than the amount of space used by the disk block addresses in a UFS inode, then the data is written directly into the inode (on-disk). A symbolic link of this type is marked by modifying the flags field (previously unused) to indicate a fast symbolic link. When such a symbolic link is used, there is no need to read any data blocks; the data is in the inode. This mechanism is implemented entirely in UFS and has no effect on the VFS or other file systems.

11.4.3 The System V File System

The System V File System in OSF/1 is derived from System V, Release 2. The VFS and vnode operations are derived from the OSF/1 UFS implementation, based on 4.3BSD Reno UFS.

This section provides an overview of the implementation of the System V File System (SysV FS) in OSF/1. It concentrates on implementation issues encountered which are specific to OSF/1, and does not cover SysV FS internals.

The SysV FS can be exported by NFS servers to be mounted over the network. OSF/1 supports paging to and from a System V File System. The System V File System has been enhanced to support BSD-style symbolic links.

11.4.3.1 SysV FS Funneling

SysV FS in OSF/1 is not parallelized. This means that it provides no multiprocessor synchronization and does not protect its data structures. As a result, on a multiprocessor system, all SysV FS operations must take place on a single processor, designated as the master processor, or **unix_master**. OSF/1 is designed to be highly parallel in a multiprocessor environment. However, for compatibility, it includes mechanisms for integrating subsystems that cannot operate correctly in a multiprocessing environment.

OSF/1 VFS and vnode architecture provides well-defined entry points into file system implementations through the VFS operations and vnode operations. The funneling mechanism of the VFS architecture allows a subsystem to be automatically funneled onto the **unix_master** processor when one of its entry points is called. The SysV FS in OSF/1 is such a subsystem. Because of the automatic funneling, the SysV FS implementation can operate as if it were on a uniprocessor system. See Chapter 12 for more information about funneling.

11.4.3.2 VFS and Vnode Implementation of SysV FS

The implementation of SysV FS in the OSF/1 VFS architecture leverages heavily upon the UFS implementation. Many of the vnode and VFS operations are quite similar. The differences are in the handling of on-disk data structures, and these operations are fairly well isolated inside the SysV FS implementation. One major area of integration is the locking of SysV FS inodes in the absence of vnode lock and unlock operations.

Several extensions to the SysV FS are included in the OSF/1 implementation. Among these are support for symbolic links, support for the **rename**( ) operation, and partial support for **truncate**( ) and **ftruncate**( ). It is also possible to map a SysV FS file using **mmap**( ), and to page to and from a SysV FS file.

11.4.3.3 Inode Locking

SysV FS is not parallelized like UFS, but it is implemented in the OSF/1 VFS and vnode architecture, which has no vnode locking operations. Because SysV FS is guaranteed to run on a single processor, there are no synchronization problems unless it sleeps, as it does when performing disk I/O. As a result, the only special locking SysV FS requires is inode locking when it performs reads, writes, or other operations that affect the file data. Access to file system metadata is provided, as it is in UFS, by the buffer cache synchronization.

SysV FS also has the same issue as UFS with regard to directory locking between the lookup and create operations. SysV FS in OSF/1 implements the same algorithm as UFS for this situation; it detects changes to the directory and performs an extra lookup if the directory has been modified between the initial lookup and the create operation.

11.4.3.4 SysV FS Extensions

The OSF/1 implementation of SysV FS allows creating and reading symbolic links. However, it does not implement fast symbolic links, as UFS does in OSF/1.

SysV FS supports the atomic renaming of files, a feature not present in the System V, Release 2 source. The implementation is based on that of UFS.

The **truncate( )** and **ftruncate( )** operations are not present in System V, Release 2. These operations have been added to SysV FS in OSF/1. In OSF/1, it is not possible to truncate a SysV FS file to any length smaller than its current size, other than zero. It is possible, however, to truncate a file to a size larger than its current size.

In order to support the execution of binaries stored on a SysV FS, the OSF/1 SysV FS supports file mapping. This allows demand-paged executables to operate correctly. It also permits SysV FS files to be used by the **mmap( )** operation, and to be used by the VM system as paging files.

The System V, Release 2 implementation of the SysV FS permits configuration of file systems with a logical block size of 512 bytes or 1024 bytes. The OSF/1 SysV FS additionally allows a logical block size of 2048 bytes.

The OSF/1 SysV FS returns **ENAMETOOLONG** instead of truncating a filename, and uses the POSIX semantics for setting the GID by using the creator's GID, not that of the parent directory.

In OSF/1, it is not possible to use a SysV FS as the root file system.

11.4.3.5 Dynamic Configuration Changes

In OSF/1, SysV FS is dynamically configurable. The configuration manager daemon calls the **sysv_fs_configure( )** routine, which calls **vfssw_add( )** to register the SysV FS's VFS operations vector in the virtual file system switch (**vfssw**).

Dynamic unconfiguration removes the operations vector from the **vfssw** by calling **vfssw_del( )**.

11.4.4 File System Security Extensions

OSF/1 supports a variety of file system security extensions, which are selected at compilation. Chapter 15 describes the security conditionals and the features they enable in detail.

11.4.4.1 Tagged and Untagged UFS File Systems

OSF/1 security configurations support two UFS file system formats: one that has been extended to include extra security attributes, and another that is intended to provide backward compatibility with existing file systems at the expense of some flexibility with respect to security attributes.

In order to take advantage of security features such as access control lists, file-based privileges, and mandatory access control, additional information must be included in the inode of each file. Because the traditional UNIX inode structure has only a limited amount of space reserved for expansion, the OSF/1 inode structure was extended to hold these additional file attributes.

File systems whose inodes include these extra fields are called extended format file systems (also known as *tagged* file systems). All of the additional fields are allocated, even though all may not be used in a given configuration. In cases where backward compatibility is more important than extra security, such as when importing file systems from another machine, it may be desirable to mount file systems that are in the more traditional OSF/1 file system format, known as the unextended format. This file system format is sometimes also referred to as an *untagged* file system.

Because the added inode fields do not exist on unextended file systems, it is impossible to set some security attributes on a per-file basis if the file resides on an unextended file system. However, the **mount()** command options can be used to specify some attributes that apply to all files on a given file system. When traditional format file systems are mounted, the attributes associated with all files in that system are stored in the mount table.

11.4.4.2 Extra Privilege Checking

Privileges control access to operating system functions by the programs that run in processes. The trusted kernel defines discrete privileges to protect functions that are reserved for the superuser in a traditional UNIX kernel. Trust is assigned to application programs through the contents of the two privilege sets associated with executable files.

Potential set The set of privileges that a program is trusted to raise

Granted set The set of privileges placed into a process's effective set when it executes the file as a program

Programs can also be designated as trusted by making them SUID to root and placing the **sucompat** privilege in the program's potential and granted privilege sets.

The privilege mechanism is defined to separate the root power into specific rights that can be individually raised and lowered to enable a privilege only for the duration of the operation for which the power is required.

Privilege sets on files appear in the on-disk and in-core inode data structures. Because privilege sets are only relevant on files that can be run as programs, privilege sets on other objects are not implemented.

11.4.4.3 Vnode Additions

The file-system-independent data structure for a file stores a separate operations vector that points to a set of operations for setting and retrieving attributes on extended format file systems. If the system supports mandatory access control, the file-system-independent file data structure also stores a flag to indicate whether the file is a multilevel parent directory.

11.4.4.4 Multilevel Directories

A multilevel directory has separate child directories for each sensitivity level, and is used to implement directories that must be accessible to processes at more than one sensitivity level. When an unprivileged process references a multilevel directory, it is automatically diverted into the child directory corresponding to the process's sensitivity level.

11.4.4.5 Superblock Changes

The on-disk superblock for extended format file systems stores a magic number that is different from the one traditionally associated with the specified file system type. The nonstandard magic number indicates to unmodified software that the underlying file system cannot be manipulated by software not prepared to deal with the changed format.

Each of these file system security extensions is described in detail in Chapter 15.

Chapter 12

Sockets

The OSF/1 operating system provides sockets as an interface to local and network communications. A socket is a communications endpoint. In user space, a socket is represented by a file descriptor, and in kernel space it is represented by a **socket** data structure. The discussion in this chapter is from the perspective of the kernel, and concentrates on the implementation of the sockets framework. It does not discuss the design of sockets or of network protocols in any detail because this material is widely available elsewhere. Instead, the text focusses on the way these elements are integrated into the OSF/1 framework. An extensive discussion of sockets can be found in *The Design and Implementation of the 4.3BSD UNIX Operating System*[1].

---

1. Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*. Reading, Massachusetts: Addison-Wesley Publishing Co., 1989.

12.1 The Socket Framework

The socket architecture provides a framework in which different communications protocols can be installed and accessed in a consistent manner. At the user level, the framework appears as the socket interface. Within the kernel, it provides a method of memory management and scheduling of protocol processing for network packets. The ability to install and initialize protocols into the kernel dynamically is an integral part of the OSF/1 framework's design.

The OSF/1 framework is fully parallelized for multiprocessor systems and can incorporate both parallelized and nonparallelized protocols. The OSF/1 operating system provides two fully parallelized domain families: Internet (TCP/IP) and UNIX IPC.

12.2 The Socket Programming Interface

Some of the system calls for manipulating sockets are

socket( ) Creates a socket

bind( ) Binds a name to a socket

connect( ) Connects two sockets

listen( ) Listens for socket connections

accept( ) Accepts a new connection on a socket

read( ) Reads data and information to a socket

write( ) Writes data and status and control information from a socket

ioctl( ) Controls a socket

close( ) Closes a socket

Information about these system calls can be found in the *OSF/1 Programmer's Reference*.

12.3 Domains and Protocols

The sockets framework must incorporate protocols from distinct domains in a consistent way. To do so, the framework must solve two problems:

- It must provide a consistent interface for making processing requests of protocols.
- On multiprocessor machines, it must provide a means for serializing access to nonparallelized protocols.

Because the kernel permits dynamic addition of protocol code, the framework must also be able to allocate and initialize all of the relevant data structures dynamically.

The **domain** and **protosw** data structures are designed to address these issues. The **domain** structure maintains per-domain data and a set of pointers to domain specific functions. The **protosw** structure functions as a switch that provides a standard interface for socket-to-protocol and protocol-to-protocol service requests. Both structures are similar to the equivalent 4.3BSD structures, but the **domain** structure has added fields for parallelization.

12.3.1 Domain Overview

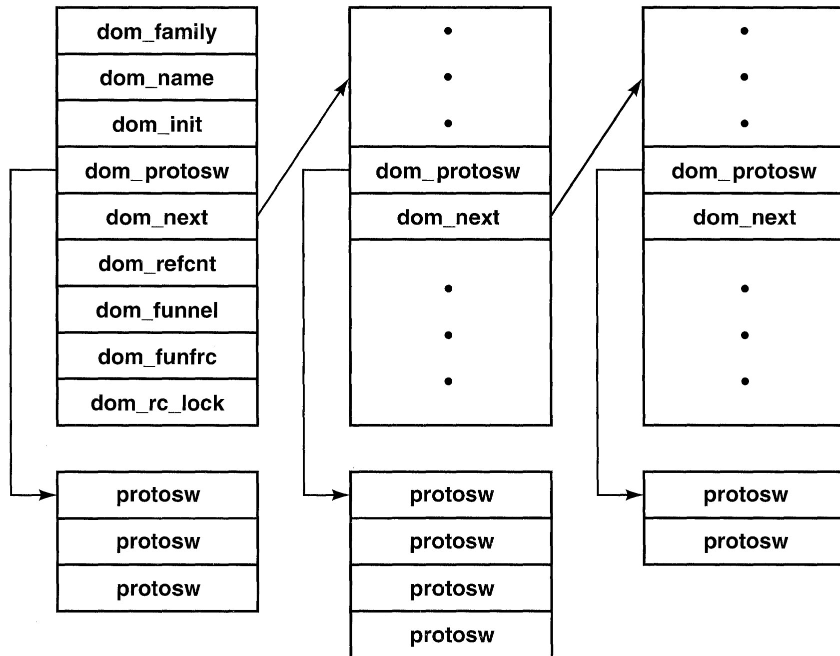
To manage domains and protocols, the kernel maintains a linked list of domain structures. Domains can be added and removed from the list dynamically.

Each domain structure references an array of **protosw structures**. Each **protosw** structure in the array corresponds to a protocol in the domain's protocol family. The **dom_protoswNPROTOSW** element indicates the end of the domain's **protosw** array and can be used as a reference for operations that need to know how long the **protosw** array is.

12.3.2 The domain Structure

The kernel allocates a **domain** structure for each domain or protocol family available for communication. Figure 12-1 shows the **domain** structure and an example of domain structures linked into a list.

Figure 12–1. The domain Structure



The **domain** structure contains the following fields:

dom_family Protocol family identifier.

dom_name Pointer to the ASCII name of the domain.

dom_init A domain initialization function.

dom_protosw
Pointer to an array of protocol switches for the protocols in this domain. The **dom_protoswNPROTOSW** indicates the

end of the protocol switch array and is used by operations that must know the size of the array.

dom_next Pointer to the next **domain** structure in the list of domains.

dom_refcnt A count of sockets and interrupt service routines (ISRs) in this domain.

dom_rc_lock
A simple lock for **dom_refcnt**.

dom_funnel and **dom_funfrc**
Functions used to serialize nonparallel protocols on multiprocessor systems.

12.3.3 Adding and Deleting Protocols

The OSF/1 operating system permits protocols to be added and removed dynamically.

The **domain_add( )** function adds protocol families to the kernel's list of domains. The function is called with a pointer to the domain structure to be added to the list. The **domain_add( )** function first checks to see whether the domain is already on the list. If the domain is found, **domain_add** returns an error. If not, **domain_add** adds the **domain** structure to the head of the domain list.

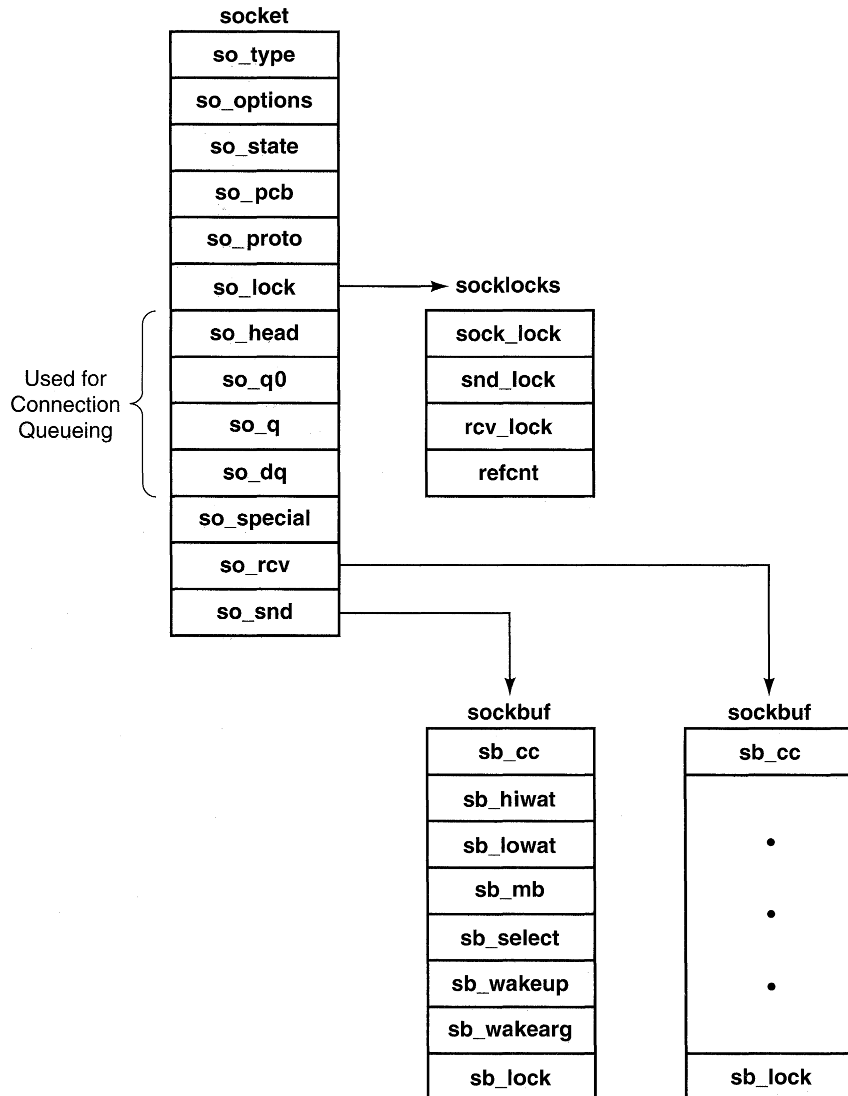
Next, **domain_add( )** initializes the domain and its protocols. It calls the domain initialization routine pointed to by the domain's **dom_init** element, and calls the protocol initialization routine pointed to by the **pr_init** element of each protocol switch in the domain's **protosw** array. It uses the **dom_protoswNPROTOSW** pointer to find the end of the array.

Each time a socket is created in a domain, or a new connection is accepted on an existing socket, the domain's **dom_refcnt** is incremented. Adding an interrupt service routine for the domain also increments the reference count. The **domain_delete( )** function deletes domains. This function checks **dom_refcnt**, returning an error if the domain is still in use by a socket or has an ISR installed.

12.4 The socket Data Structure

Within the kernel, sockets are referred to through the **socket** data structure. The socket data structure is shown in Figure 12-2.

Figure 12–2. The socket Data Structure



The socket structure includes the following fields:

so_type Holds the socket's type; for example, **SOCK_STREAM**, **SOCK_DGRAM**.

so_options Holds a flag that indicates options specified by the **setsockopt( )** system call.

so_state Holds a flag that indicates the socket's state.

so_pcb Points to a *protocol control block* for this socket.

so_proto Points to the protocol switch for the protocol used by this socket. The protocol switch, in turn, points to the domain.

so_lock Points to the locking structure used to serialize access to the socket buffers.

It also includes several fields used for connection queuing:

so_head Points to the socket that accepted the connection for this socket.

so_q0 and **so_q**
Anchor queues of partial connections and connections ready to be accepted. The socket also includes counts of queue members and a **so_qlimit**, the maximum number of connections that may be queued at a socket.

so_dq A queue of connections that need to be freed. This field is used for shutting down a socket that is accepting connections on a multiprocessor system. The queue enforces orderly cleanup of connections in order to avoid violation of the lock hierarchy when multiple threads are aborting and freeing connections.

so_special Holds a set of special state bits. These include a bit to indicate whether the socket is in a parallelized domain and thus lockable (**SP_LOCKABLE**) and a bit to indicate that the socket should be freed when unlocked (**SP_FREEABLE**).

The two **sockbuf** structures, **so_rcv** and **so_snd**, are used to queue incoming and outgoing **mbuf**s at the socket. Section 12.7 discusses **mbuf**s. Each **sockbuf** contains the following fields:

sb_cc A count of bytes in the buffer.

sb_hiwat and **sb_lowat**
The high and low water marks for the buffer. **sb_hiwat** gives the maximum allowable size of the queue. Threads blocked waiting for space in the queue are awakened when the size drops below **sb_lowat**.

sb_mb Anchors the chain of **mbuf**s queued at the buffer.

sb_select A queue of threads selecting on this socket. This field is used by the OSF/1 select implementation. When an event occurs, the select event for the threads on the queue is posted, notifying the threads.

sb_wakeup Points to an alternate wakeup routine. The routine is called with an argument that holds new socket state and the arguments pointed to by **sb_wakearg**. XTI uses this to invoke **qenable**( ) for queue enabling; for example, when an XTI socket wakes up. The NFS socket code also uses this field.

sb_wakearg An argument to be passed to an alternate wakeup routine.

sb_lock A lock that is not currently used, but which may be used in the future to implement a finer granularity of locking for socket buffers on multiprocessor systems.

12.5 Scheduling Network Activity

The sockets framework is usually configured to handle network processing requests with a set of interrupt service routine (ISR) threads. Despite their name, and even though they act in the service of a hardware interrupt, these threads do not run at a hardware interrupt level, or even at a software interrupt level. The thread package minimizes the amount of processing that is actually done in interrupt context. The network interface schedules the processing request from interrupt context, but the actual request processing is done in the context of an ISR thread.

The framework can also be configured to handle packets in software interrupt context. However, because the socket lock cannot be taken in interrupt context, the uniprocessor configuration must be used. This also means that certain memory and **mbuf** allocations may not be allowed to

block, making them less reliable. The performance of the system configured with software interrupts is comparable to one with threads.

12.5.1 Event Management

Protocols and frameworks register events by calling the **netisr_add()** routine. Examples of events are packets received, regular timers, such as the protocol fast and slow timeouts, and events requiring deferred execution, such as freeing of memory.

The routine that calls **netisr_add()** selects a manifest constant to describe the event, which will be used by **schednetisr()** to schedule processing when the event occurs. This constant may be specified for "well-known" events, or it may be dynamically selected by **netisr_add()** for use on a subsystem internal basis. The only requirement is that the constant be known to the source of the event.

When the event occurs, the **schednetisr()** macro is invoked. It may be called from **timeout()**, from a network event, such as a packet being received, or from the kernel; for example, when a request is made to free memory.

The call to **schednetisr()** is implemented as a macro. It increments the **pending** element of the protocol's **netisr** and then wakes up an ISR thread waiting on the **netisr_thread** event. The woken-up thread eventually runs, calls **Netintr()**, and then blocks to await further network requests. (In a kernel configured to handle events in software interrupt context, **schednetisr()** schedules a software interrupt instead of waking up a thread. **Netintr()** handles the interrupt.)

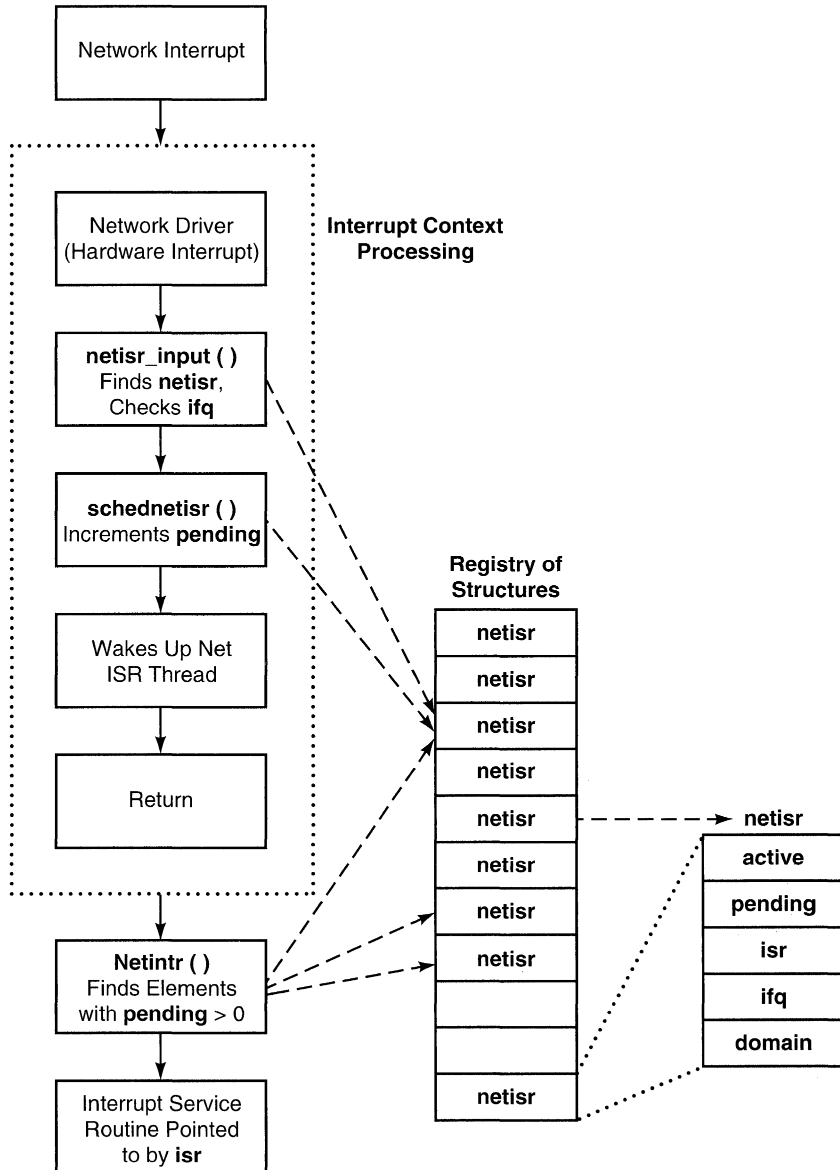
Netintr() finds **netisr** structures with pending requests (**pending** > 0). When it finds one, it increments the **active** member and sets the **pending** member to 0. **Netintr()** then calls the interrupt service routine pointed to by **isr** to perform the processing required by the protocol. The active member ensures that a registered **isr** cannot be deleted while it is active, while the domain reference count, **dom_refcnt**, ensures that a domain cannot be deleted while an ISR is installed that references it.

When entering the domain code (through the call to the domain's interrupt service routine), **Netintr( )** uses the domain funnel mechanism to ensure that the appropriate synchronization takes place. This mechanism allows nonparallelized protocols to operate within the OSF/1 parallelized framework. The domain funnel is described in Section 12.6.5.

12.5.2 The netisr Structure

The kernel maintains the information required to schedule protocol activity in a registry of **netisr** data structures. (See Figure 12-3.) This registry maintains a set of **netisr** structures for each protocol family to handle events specific to that family. The registry also contains a **netisr** structure for packets not directed to a specific protocol (wildcard packets) and a structure that is used by the **mbuf** allocation mechanism, plus members for timer events, STREAMS events, and so forth.

Figure 12–3. Managing Network Interrupts



The elements of the **netisr** structure are as follows:

active Counts the number of active invocations of the protocol's **isr**.

pending Counts pending requests for the protocol's **isr**.

isr Points to the ISR that the protocol uses to handle events such as incoming packets.

ifq Optionally points to the protocol's input queue. Incoming packets are placed on this queue for processing by the protocol.

domain Optionally points to the **domain** structure for this protocol stack. This is used by the domain funnel mechanism (see Section 12.6.5). The domain reference count, **dom_refcnt**, is incremented for each **netisr** that references it.

12.5.3 Packet Processing

To request protocol processing, a network driver calls the routine **netisr_input**( ) with

- A constant that identifies the protocol that should receive the packet
- A pointer to the **mbuf** holding the packet

The routine uses the constant to find the correct **netisr** in the array of **netisr** structures. It then checks any queue pointed to by the **netisr**'s **ifq** to see if there is room for another packet. If not, the packet is dropped and the appropriate error statistic is incremented. If there is room, the packet is queued and **netisr_input**( ) then calls **schednetisr**( ) to schedule protocol processing of the packet. It is possible to register a "wildcard" **isr** to receive a copy of all packets. When such an **isr** is registered, **netisr_input**( ) enqueues a copy of each packet in the same manner. Additionally, it is possible to register an "other" **isr** to catch packets that would otherwise be discarded; that is, packets that are destined for an unregistered **isr**.

12.5.4 The isr Threads

At initialization time, the networking code starts a number of interrupt service threads. These simply loop, calling **Netintr( )** and blocking on the event **netisr_thread**. For efficient processing of network activity, it is essential to have a thread available when it is needed. Because threads can block waiting for resources, the system generally establishes a few more threads than processors.

Other events, such as fast and slow timeouts for protocols, are also registered as **netisr** structures to be handled by these threads. On each cycle, these threads go through the list of domains, calling the fast and slow timeout routines for the protocols of each domain. Other timers are registered by subsystems such as ARP.

12.6 Synchronization

The OSF/1 networking subsystem can be configured for uniprocessor or for multiprocessor environments through compilation switches. The networking framework is fully parallel on multiprocessor systems. The TCP/IP and UNIX IPC protocol stacks have also been fully parallelized. However, the framework can also incorporate nonparallelized protocols on multiprocessor systems using the domain funnel mechanism. The optional XNS protocol stack is an example.

Multiprocessor networking code presents two essentially orthogonal synchronization problems when accessing common data structures.

- As with uniprocessor code, some data structures (for example, packet queues) may be accessed either from the current thread context (from above) or from interrupt context (from below). When such a data structure is accessed, the interrupt level must be raised to prevent corruption of the data by simultaneous access from below.
- Common data structures may also be accessed simultaneously—either from above or below—by different processors. The networking code must synchronize access by either forcing execution onto a single processor or locking the data structures. The fully parallel network code

employs locks for synchronization. When accessing nonparallelized protocols the networking code may also force execution onto a single processor using the domain funnel mechanism, described in Section 12.6.5.

For data structures that may be accessed from below, the networking code must both raise the interrupt level and either force execution onto a single processor or employ locking. Failure to raise the interrupt level before taking a lock could leave an interrupt service routine spinning forever on the lock.

12.6.1 Locking

The networking code employs a set of locks to serialize access to common data structures such as sockets, socket buffers, and queues. In the OSF/1 networking code, locks occur mainly around the network interface. For example, locks protect the interface queues used to pass packets from the network interface to the protocols. These locks are always taken with the interrupt level raised to block further interrupts from the network.

In uniprocessor implementations or with software interrupt-based ISRs, locking is normally turned off. *Spl* synchronization—raising the interrupt level—is used to serialize access to data structures that may be accessed from interrupt context.

12.6.2 Socket Locks

Socket locks are a basic element in the parallelization scheme of the OSF/1 networking code. The socket framework provides socket locks to serialize access to sockets. Parallel protocols use socket locks to synchronize with the socket layer.

Sockets pose two kinds of serialization problems:

- Access to socket members, such as the **sockbuf**, must be synchronized.
- Sockets may need to be linked to and unlinked from other data structures, such as **pcb** (process control block) structures, which may also be locked. Locking must be carefully coordinated among structures to avoid deadlocks.

Socket locking is controlled by the **socklocks** structure contained in each socket, as illustrated in Figure 12-2. Currently the only elements used are **sock_lock** and **refcnt**. Because operations on the socket buffers are very frequent, it would be feasible to provide finer locking granularity by locking the buffers individually. In practice, the coarser locking provided by a single socket lock works well and greatly simplifies coding and deadlock avoidance. The **SOCKET_LOCK** macro, which takes a write lock on the **sock_lock** element, is used in code that locks sockets.

The **soreceive( )** routine illustrates typical socket locking actions that networking code should take in order to serialize access to the socket receive buffer:

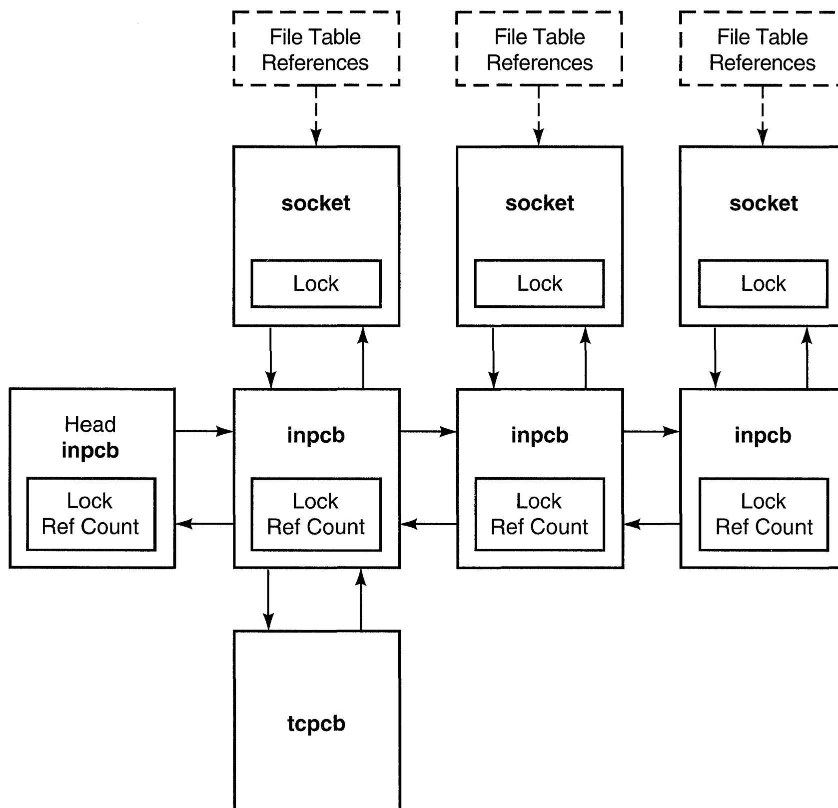
1. Acquire the socket lock.
2. Check the receive buffer for data. If there is not enough, call **sosbwait( )**, which may wait for more. If **sosbwait( )** decides to wait, it calls **sosleep( )**, which releases the socket lock before sleeping.
3. When **sosleep( )** wakes up, it reacquires the socket lock. The lock remains held as **soreceive( )** works its way through the **mbuf**s queued at the receive buffer, assembling the requested data.
4. Copy data out to user space or pass the **mbuf** chain directly up to the caller. (The latter method is used by the NFS and the XTI streams-to-sockets interface.) While copying out, the socket lock is released again. This avoids holding the socket lock during a possible page fault and allows more data to be queued concurrently. When the copy is done, the lock is acquired and the socket and sockbuf are rechecked and updated.
5. Release the socket lock.

Meanwhile, data may arrive for the socket as a result of network activity. For example, the TCP protocol may call **tcp_input( )** to put some **mbuf**s on the socket's receive queue. Once the protocol is ready to process the packet and has found the socket (by looking through the **in_pcb** chain), it reacquires the socket lock. The protocol then processes the packet, for example, by queuing the **mbuf** chain at the socket receive buffer with a call to **sbappend( )**. When all processing is done, the socket lock is released.

12.6.3 Internet Domain Locks

Sockets are linked to **pcb** structures, which must also be locked to serialize simultaneous access. Figure 12-4 illustrates locks in a socket using TCP/IP.

Figure 12–4. Internet Domain Locking



The socket is linked to an **inpcb**, which is itself linked to a per-protocol **pcb**, in this case a **tcpcb**. The **inpcb** is part of a doubly linked list of all the **inpcb**s for TCP sockets. This list is headed by a header **pcb**. To establish connections and create or tear down sockets, the networking code must be carefully constructed to take and release locks in the right order and hold them for the correct amount of time.

For sockets connected with Internet protocols, the following elements are used:

- A lock on the **inpcb** associated with each socket
- A reference count for each **inpcb**
- A lock on the **pcb** that heads the per-domain chain of **inpcb**s
- A lock on each socket

There is also a reference count on each socket lock, but for Internet sockets this is always 1. Because sockets may be referenced by file table entries, it is also necessary to keep track of the references per socket. However, this information is held elsewhere; for example, in the file table.

The overall locking strategy is as follows:

- The socket lock is always taken ahead of and released after the **inpcb** lock. Failure to observe this hierarchy can lead to deadlock.
- The **inpcb** lock is used to protect each connection. Any per-protocol **pcb** is protected implicitly by locking the **inpcb** to which it is linked.
- The lock on the head of a per-protocol chain of **inpcb**s protects the chain for the addition or removal of **inpcb**s. One thread holding an individual **inpcb** lock does not prevent another thread from manipulating the **inpcb** chain.
- When a socket and **inpcb** are created, the **inpcb** reference count is set at 1. This reference is dropped when the protocol **PRU_DETACH** routine is called, either by the protocol or from **soclose( )**.
- When an **inpcb** reference count goes to 0 (zero), the **inpcb** is deallocated and **sofree( )** is called.
- When an **inpcb** is looked up; for example, when a packet is received, a read lock is taken on the head of the per-protocol chain. A reference is taken on the **inpcb** and the **inpcb** lock is acquired. The reference is then decremented. If the reference count drops to 0 (zero), the lookup fails.
- The existence of the **socket** is protected by two references, the file structure from above, manifested by a false **SS_NOFDREF** bit, and by a non-NULL **so_pcb** from below. The **soclose( )** routine must be called to assert **SS_NOFDREF** before the socket can be destroyed. **soclose( )** calls the protocol's **PRU_DETACH** entry to attempt to discard the procotol control block. The protocol may elect to do this later, however,

in which case **soclose()** returns without destroying the socket. The protocol is then responsible for doing so at a later time.

Sockets are destroyed through a call to **sofree()**. The **sofree()** call itself does not unlock and free the socket, however. Instead, it checks that **so_pcb** is NULL and **SS_NOFDREF** is set, and then it sets the **SP_FREEABLE** bit in **so_special**. When the caller of **sofree()** later calls **sounlock()**, the socket is destroyed. In this way, either the protocol or the socket layer can call **sofree()** whenever each is done with the socket, without having to perform any further synchronization.

12.6.4 UNIX IPC Socket Pairs

When two sockets are directly connected as a socket pair, they share the same lock in order to avoid race conditions and deadlock. In this case, the socket lock reference count is incremented to 2. When two sockets are connected unpaired, the code follows a careful socket lock and unlock order to ensure that no deadlock results.

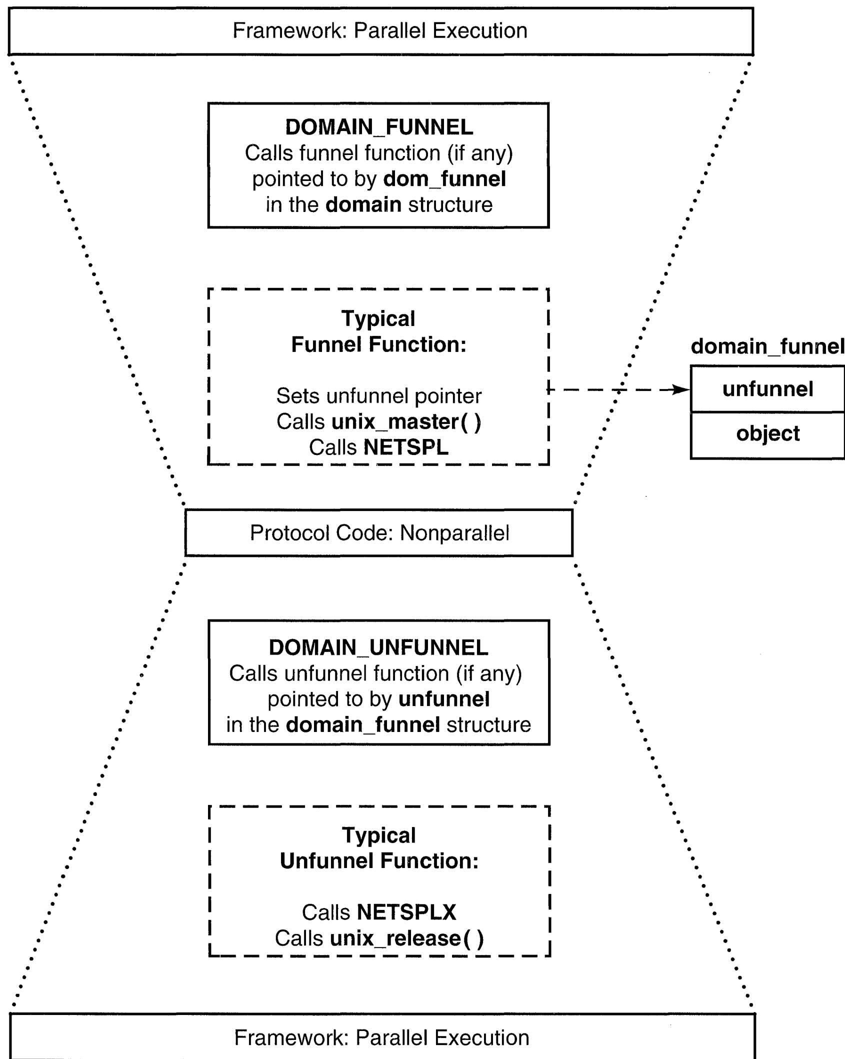
12.6.5 The Domain Funnel

The socket framework does not assume that all protocol stacks have been parallelized with socket and other locks. A nonparallelized protocol may assume that it is executing on a single processor and provide only spl synchronization. The domain funnel mechanism provides a general means of forcing execution onto a single processor or carrying out other serialization operations before calling into protocols that are not fully parallelized. In effect, the kernel funnels execution across boundaries between parallelized and nonparallelized code. The funnel appears in essentially the same places in the OSF/1 framework as **splnet** synchronization appears in nonparallelized BSD.

The domain funnel mechanism has three important elements:

- The domain funnel macros, such as **DOMAIN_FUNNEL**, **DOMAIN_UNFUNNEL** and others, are defined to declare the necessary domain funnel data structures and to call the domain-specific funnel and unfunnel functions. Each domain can specify a domain funnel function to provide the specific serialization that the domain requires. Generally, this function forces execution onto the UNIX master processor and raises the interrupt level. It may also provide more complex serialization procedures.
- The **dom_funnel** element of the domain structure points to the domain's funnel function. Fully parallel domains set the **dom_funnel** pointer to **NULL** since they require no funneling.
- A **domain_funnel** structure is used for bookeeping and control of each funnel. The structure is shown in Figure 12-5.

Figure 12–5. The Domain Funnel



The **unfunnel** element points to a function that undoes the serialization initiated by **dom_funnel**. The **object** element can be used for bookkeeping and control at the funnel, holding, for example, the previous interrupt priority to be passed to **splx**( ) by the unfunnel element.

Code that uses the funnel mechanism declares a **domain_funnel** structure at each point where a funnel is required. The code then includes the **DOMAIN_FUNNEL** macro, which results in a call to the funnel function pointed to by the domain's **dom_funnel** field. To terminate the funnel, the code includes a **DOMAIN_UNFUNNEL** macro, which translates to a call to an unfunnel function pointed to by the local funnel structure's unfunnel element.

A typical domain funnel function first sets the unfunnel pointer in the local **domain_funnel** structure to a matching unfunnel function. It then calls **unix_master**( ) to force execution onto the master processor. Finally, it calls the **splnet**( ) function to enforce the appropriate level of spl synchronization. The matching unfunnel function undoes the funnel by calling the **splx**( ) function and then releasing the UNIX master processor. Both functions may use the value of the unfunnel pointer in the local funnel structure for sanity checking to make sure the funnel has not been reentered.

The domain funnel macros are present in the networking code wherever it must call across the protocol stack boundary. They are used frequently by sockets. For example, when **socreate**( ) calls a protocol's **usrreq**( ) function to attach a protocol to a new socket, it uses the domain funnel macros to assure that serialization takes place for protocols that are not fully parallelized. The domain funnel macros are also present in **Netintr**( ) where it calls up into the protocols.

12.7 Memory Management

OSF/1 uses **mbuf**s and **clusters** for memory management within the socket framework.

Network data is generally held for a short (or at least limited) length of time and occurs in widely varying amounts. The allocation scheme for buffers should be able to provide

- Memory in units of widely varying size
- Storage for the relatively short amount of time that it takes to pass data through the networking subsystem

- Allocation units linked together in lists and in chains of lists according the requirements of the protocol handling the communication

The **mbuf** mechanism meets these requirements.

12.7.1 Mbufs and Clusters

An **mbuf** is an allocation unit capable of storing a limited amount of data internally or of referencing an area of external storage. The **mbuf** structure contains pointers to allow **mbuf**s to be chained together. In general, the system treats an **mbuf** chain as a single allocation unit. Chains may themselves be linked together into lists. For example, a communications protocol may link a sequence of network packets (represented as **mbuf** chains) into a list. In general, datagram or record-oriented protocols use lists of **mbuf**s while stream-type protocols, such as TCP, use a single chain.

As data arrives from the network or is generated by user threads, it is passed through the layers of the system as chains of **mbuf**s. Sockets move data from above (from user space, for instance) into chains of **mbuf**s and pass them to the protocols. The protocols process them and pass them down the protocol stack until they are sent by the network interface. Similarly, as packets arrive from the network, they are assembled into chains of **mbuf**s and passed up through the protocol layers to the socket layer.

There are, effectively, two sizes of **mbuf**s, regular and large. The regular **mbuf** consists of a header and a data area, and can usually hold about 100 bytes of data. The large size consists of an **mbuf** that has been expanded by adding a cluster, which is a data element that is typically much larger than an **mbuf**. The size of a cluster is configurable at build time as a machine-dependent parameter. The manifest constant **MCLBYTES** defines the size of a cluster. There is a third kind of **mbuf**, one with arbitrary external data, which is described later.

Every network protocol adds or deletes prefix information as data traverses the layers of the protocol. As a design feature of **mbuf**s, protocol prefixes are inserted or removed by inserting or deleting **mbuf**s at the head of the chain, or by adjusting the **m_data** pointer.

12.7.2 The mbuf Data Structure

Each **mbuf** begins with a header structure, **m_hdr**, which includes the following fields:

m_next Points to the next **mbuf** in a chain.

m_nextpkt Points to the next **mbuf** chain in the list of chains.

m_len Gives the amount of data held in the **mbuf**.

m_data Points to the location of data in the **mbuf**.

m_type Indicates the type of data in the **mbuf**, such as simple data, a packet header, or some other type. The value **MT_FREE** indicates that the **mbuf** is not in use.

m_flags Indicates the type of the **mbuf** data. The value **M_EXT** indicates that the **mbuf** has external storage, and the value **M_PKTHDR** indicates that it has a packet header.

The rest of the **mbuf** holds internal data and/or other structures and pointers. When the **mbuf** is the first **mbuf** in a chain, this area often contains a **pkthdr** structure that gives the length of data in the chain. When the **mbuf** has external data, this area also contains an **m_ext** structure with the following fields:

ext_buf Points to the external data buffer. This may be a cluster allocated through **malloc( )**, or it may be some other area of memory.

ext_free Points to a routine that is used to free the external buffer. This is used when the external buffer was not allocated through **malloc( )**. A NULL value indicates a normal cluster.

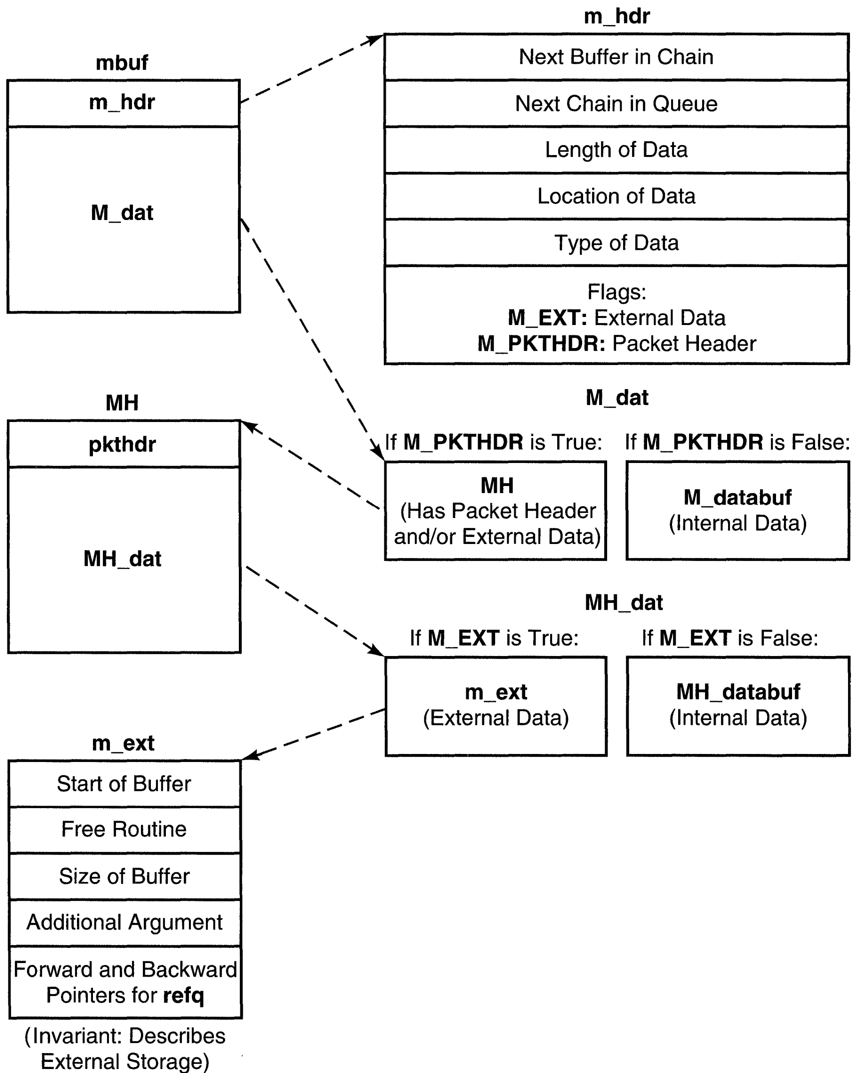
ext_size Gives the size of the external buffer.

ext_arg An additional argument passed to **ext_free**.

forw and **back**
Two pointers, used to maintain a linked list of **mbuf**s that use the same external storage buffer. This allows copying by reference and permits the system to keep track of the external buffer as it frees the associated **mbuf**s.

Figure 12-6 shows the **mbuf** data structure.

Figure 12–6. Components of the mbuf Data Structure



12.7.3 Allocating mbufs

The basic **mbuf** allocation strategy is to try to get the entity required for the least cost. Allocation for **mbuf**s and for clusters uses the following sequence:

1. Try to allocate the memory with **malloc( )**. This call is made with **malloc()**'s **M_NOWAIT** flag set to inhibit blocking. If the allocation is successful, return the memory after performing any initialization.
2. If **malloc( )** was unsuccessful, check the **M_WAIT** flag passed from the caller. If it is false, return failure immediately. This method is normally used by drivers allocating memory at interrupt time; for example, when receiving a packet.
3. Try to recover some memory from the protocols by using any **pf_drain( )** routines exported in their domain structures. This method will free any "expendable" memory, such as packet fragments that can be recovered in retransmissions.
4. Call **malloc( )** with the **M_WAITOK** flag set. The **malloc( )** call will block until memory is available. Initialize the returned memory and return.

Several macros are available to allocate **mbuf**s and clusters, including the following:

- **MGET** allocates an **mbuf** initialized for internal data. Similarly, **MGETHDR** allocates an **mbuf** initialized as a packet header.
- **MCLGET** adds a cluster to an **mbuf**. The **MCLALLOC** macro is used to allocate a cluster, and the **mbuf m_ext** structure is initialized appropriately. The **M_EXT** bit in **m_flags** is set on success. **MFREE** frees an **mbuf** and any associated external storage.

12.7.4 External Data

Typically, external data attached to an **mbuf** is in the form of a cluster. However, arbitrary external data may be attached as well. The external storage pointed to by **m_ext.ext_buf** may reside anywhere in the system's address space. For example, if addressable memory can be allocated on an Ethernet board, an **mbuf** can access it directly, eliminating the need to copy

data from the board.

This technique is used to pass data back and forth between the XTI/Streams module and the socket layer. The routines **mbuf_to_mblk** and **mblk_to_mbuf** convert **mbuf**s to **mblk**s and vice versa. As much as possible, these routines avoid copying the data by mapping **mblk** external data into the **mbuf** using **m_ext** and vice versa using a similar mechanism.

Freeing an **mbuf** with this kind of external data is somewhat more complicated than freeing an ordinary **mbuf**. To maintain a record of references, **mbuf**s that reference an external area are linked together with pointers in their **m_ext** structures. When the last such reference is eliminated, the external data area can be freed.

Arbitrary external data cannot be freed in the same way as standard clusters. When **m_ext.ext_free** is NULL, clusters are freed using the macro **MCLFREE**. Otherwise, **m_ext.ext_free** points to a routine that is to be used to free the external data. It is important that the unknown free routine be called from a safe context. When **mfree** finds such an external free routine, it places the **mbuf** on a queue of **mbuf**s that require later freeing and schedules an **mbuf** event. When the **mbuf** event handler subsequently runs, it frees the external storage of any waiting **mbuf**s by calling their external free routines.

12.8 Sockets Security Extensions

OSF/1 can be configured with sockets security extensions that support mandatory access control and allow a trusted server process to receive the security attributes of clients with client requests. The socket extensions have been implemented only on UNIX domain sockets, not on Internet domain sockets. Chapter 15 describes security extensions.

Chapter 13

The OSF/1 STREAMS Framework

OSF/1 provides a STREAMS framework for the implementation of communications services. This framework consists of kernel resources and routines that can be used to create device drivers, terminal handlers, networking protocol suites, and other networking facilities. Because functions can be coded into separate STREAMS modules, developers can write kernel and applications programs that are highly portable and easily integrated into other STREAMS-based systems.

The main components of OSF/1 STREAMS are

- Data structures, declared constants, macros, and other kernel resources, which developers use for writing *STREAMS modules* and *drivers*.
- The *stream head*, a set of routines and data structures that provides an interface between user processes and the *streams* constituting communications paths. In OSF/1, the stream head also contains a special set of data structures and synchronization routines that enable streams to operate in a multithreaded environment.
- Utilities that perform functions such as stream queue scheduling and flow control, memory allocation, and callback requests.

OSF/1 STREAMS is source-code compatible with the AT&T System V Release 4 STREAMS specification. For this reason, this chapter describes

only the extensions that have been made to AT&T STREAMS. For more information, refer to the following documents:

- *UNIX System V Release 4 Programmer's Guide: STREAMS*[1]
- *UNIX System V Release 3.2 Streams Programmer's Guide*[2]
- *UNIX System V Release 3 Streams Primer*[3]

13.1 Overview

STREAMS is a set of system calls, kernel routines, and kernel resources for implementing I/O functions in a modular fashion. Modularly developed I/O functions allow applications to easily build and reconfigure communications services. An example is a module for a terminal driver that implements a terminal emulation protocol. The emulation can be turned on by adding the module or turned off by removing it, or a different emulation can substitute another module.

To communicate using STREAMS, an application creates a stream by opening a device. A stream is a full-duplex communication path between a user process and the device driver. Every stream has at least two parts, the stream head at the top and a driver (for example, a hardware driver) at the bottom. Optional modules can be inserted to process the data being passed along the stream.

Messages are the vehicle for all information passed between the stream head and the modules and driver. The stream head transfers data from a user process to the kernel and sends it "downstream" in a series of messages. The driver performs the complementary function from the device. When the stream head receives messages sent "upstream" from the driver, it makes the data available to the user process.

---

1. *AT&T UNIX System V Release 4 Programmer's Guide: STREAMS*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1990.
2. *AT&T UNIX System V Release 3.2 Streams Programmer's Guide*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1989.
3. *AT&T UNIX System V Release 3 Streams Primer*, 1986.

Modules and drivers contain pairs of data structures called *queues* that reference messages. Each module or driver also contains functions for manipulating messages. One queue handles messages in the downstream, or "write," direction towards the driver, while the other queue handles the upstream, or "read," direction toward the stream head. Each module's read or write queue is linked to the next module's read or write queue, defining the stream from stream head to driver, and from driver to stream head. Figure 13-1 is an illustration of a simple stream.

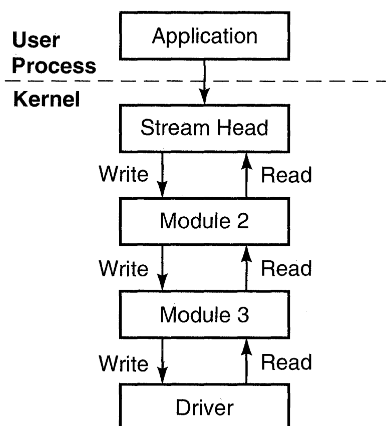
The stream head acts as an interface between the stream and the user process, providing translation from stream-based device semantics to UNIX semantics exported by the kernel. The application sees the stream as a character special file, which it manipulates with system calls such as **open( )**. The stream head interprets a standard subset of system calls, translates them into messages, and sends them along the stream.

A module or driver receives the messages, interprets them, and performs the requested operations. It returns data and control information to the user process by packaging it into messages that it sends to the stream head. The stream head then transforms the data into the appropriate form of return for the system call that was made.

Modules are pushed onto a stream or popped from it in a stack-like way. Pushed modules are inserted just below the stream head and above all other modules already on the stream. The module just below the head is always the one that is popped. Only modules that have been configured into the system can be pushed onto a stream. Different systems may support varying collections of modules.

Figure 13-1 shows the typical flow of control in a stream.

Figure 13–1. Flow in a Stream



13.2 The STREAMS Programming Interface

OSF/1 STREAMS uses the general UNIX calls that allow applications to open, manipulate, and close device files:

open() Opens a stream.

read() Reads from a stream.

write() Writes to a stream.

ioctl() Controls a stream.

poll(), select()
Notifies the application when selected events occur on a stream.

close() Closes a stream.

It also provides four STREAMS-specific calls:

getmsg(), getpmsg()
Receives a message from a stream.

putmsg(), putpmsg()
Sends a message to a stream.

For information about each of these calls, see the *OSF/1 Programmer's Reference*.

13.3 STREAMS Operations

Operations on streams proceed from system calls to the stream, and from there may complete at the stream head or result in further action and messages flowing on the stream.

13.3.1 STREAMS as a Device Driver

OSF/1 STREAMS is implemented as a device driver, which in turn manages many different STREAMS devices according to their *major number*. This implementation makes the STREAMS subsystem highly modular and minimizes dependencies throughout the system. User code accesses STREAMS through the normal control path from the system call entry points mentioned in Section 13.2.

As a device driver, STREAMS has only six entry points, as defined by the standard **cdevsw** table. These are

- **pse_open**( ), **pse_close**( )
- **pse_read**( ), **pse_write**( )
- **pse_select**( )
- **pse_ioctl**( )

The OSF/1 kernel uses **pse_select**( ) for both **select** and **poll** operations. The **getmsg**( ), **getpmsg**( ), **putmsg**( ), and **putpmsg**( ) calls are implemented as special STREAMS **ioctl**( ) calls.

STREAMS exports its own entry points for use by modules and drivers, and also registers itself into other kernel subsystems such as the **netisr** framework, the interrupt dispatcher, and so forth.

13.3.2 Flow of Control Basics

Flow of control at the stream head can be divided into two phases corresponding to the two levels of stream head functionality: the *system* and *stream* levels.

The system level can be thought of as the "top half" of the stream head; that is, the portion that interfaces to the OSF/1 kernel (the system).

The stream level is the "bottom half" of the stream head, which implements the STREAMS semantics for modules and drivers, and provides the standard STREAMS kernel utilities library.

Incoming requests from applications to particular streams are routed to the system-level routines, through the device's vnode and **cdevsw** entry. The following series of actions then occurs:

1. The stream head for the stream is located.
2. The request is interpreted and turned over to a handler routine, which places the request on a per-stream **read( )**, **write( )**, or **ioctl( )** request queue.
3. A lock is taken on the stream to obtain synchronization. The level of synchronization depends on the type of request. All **read( )** and **write( )** requests execute fully in parallel, while **ioctl( )** requests must be executed serially in order to satisfy STREAMS semantics. **open( )** and **close( )** always execute serially.
4. The handler routine then either completes the request at the stream head and returns, or builds a message and sends this message downstream for further action. If an acknowledgement or other kind of return message is expected (a situation only true for **ioctl( )** requests), the handler routine goes to sleep and waits for it. The handler may also sleep when sending data on a stream that is flow controlled.

At the stream level, the stream head receives upstream messages through its read put procedure. The stream head acts on each message according to its type; for example, it queues data or sends a signal. When appropriate, it issues a wakeup to waiting threads.

13.3.3 Stream Head Routines

Every STREAMS device is configured into two tables: the **cdevsw**, which registers UNIX character devices, and the **dmodsw**, which registers all STREAMS devices. The **cdevsw** entries for all OSF/1 STREAMS devices are identical; they contain vectors for the system-level STREAMS routines mentioned in Section 13.3.1. These routines dispatch into the stream-level routines by using information located in the **dmodsw**.

The **cdevsw** table also contains the clone device. The clone device permits any character driver to manage its minor number allocations. Many STREAMS devices can act as clonable devices. For example, a driver can implement XTI endpoints as separate streams, each with its own minor device number. The minor devices will be dynamically allocated and deallocated as endpoints are opened and closed. The STREAMS clone **open( )** operation is described in Section 13.7.

STREAMS modules are configured into a single, similar table: the **fmodsw**. Modules are not referenced by the **cdevsw** because they are accessed through an **ioctl(I_PUSH)** call on an open stream. The system-level STREAMS routines locate and use **fmodsw** entries in the same way they use **dmodsw** entries for drivers.

13.3.4 Operating System Requests

The stream head encapsulates an application's request (such as read, write, and so forth) into an operating system request structure (OSR). This structure contains everything needed to reference the stream, the request itself, and the application. Each active OSR is placed on the appropriate list (an OSRQ) associated with the stream head for the duration of its processing. The system-level and stream-level routines together perform the actions required to process the request.

The OSR serves as the link between the system-level and stream-level components of the stream head. It is used for synchronizing, serializing, and processing requests, and returning completion status to the application.

13.4 Scheduling and Flow Control

OSF/1 STREAMS schedules its activity through the **netisr** framework, which is described in Chapter 12. Only three events are registered; the first is the most frequent. These events are

NETISR_STREAMS
Scheduled each time a queue's service procedure is to be run. This occurs due to flow control or the **qenable( )** call.

NETISR_STRTO
Scheduled each time a STREAMS timeout has expired. Timeouts are requested by a **timeout( )** call by a module or driver.

NETISR_STRWELD
Scheduled in response to a weld or unweld request by a module or driver. These requests are discussed in Section 13.8.

As with sockets, STREAMS executes by default in kernel thread context under the **netisr** framework. It may also be run in a software interrupt configuration. In either case, handling of STREAMS events is entirely asynchronous and executes in parallel with and independently from other system activity.

Because the sockets framework also schedules its events through the **netisr** framework, this sharing offers advantages to drivers that bridge the two, such as XTI. When a STREAMS event delivers data to a socket, or a sockets event delivers data to STREAMS, the **netisr** framework often avoids additional context switching because the two share a common scheduler. This can greatly improve the performance of such subsystems.

13.5 Synchronization

OSF/1 STREAMS provides an unusual mechanism for implementing resource synchronization in a multithreaded environment. The original STREAMS specification provides no such mechanism; it only protects against interrupts with **splstr( )**. This method is adequate for a uniprocessor implementation, but it does not work on a multiprocessor or in a preemptive environment, and requires spending significant execution time at hardware interrupt levels. The OSF/1 STREAMS implementation rectifies all these drawbacks.

The system could protect queues by locking them during thread access, forcing all other threads to wait their turn for access, but it would behave poorly under this kind of scheduling constraint. An attempt to access the next queue could fail not only because of the familiar flow control mechanism, but also because the next queue could be locked by another thread. The result would be much more frequent rescheduling of service procedures and a corresponding degradation in stream throughput. This method would also require recoding of many STREAMS modules and drivers, especially those that do not provide service procedures.

Instead of using conventional locking, OSF/1 STREAMS grants accesses to resources in a way that maximizes execution throughput. It takes advantage of a particular feature of the STREAMS specification: the stipulation that the context of processing in a STREAMS module or driver is indeterminate except at open and close. In effect, normal STREAMS processing must act as if it were void of context during its execution, as if in response to an interrupt. The particular problem of context in open and close procedures becomes a special case, allowing the normal access of a stream to achieve maximum throughput.

The term "resource" in the remainder of this chapter describes any body or system of data that can be accessed as a whole within a stream. The most common example of a resource is a STREAMS queue. Resources are sometimes shared among queues, modules, drivers or the system. This is discussed in detail in Section 13.5.3.

An OSF/1 STREAMS *synchronization queue* is a linked list of callbacks of an access of a resource, plus a small amount of state information, consisting of locks and ownership. To acquire a resource, a thread first encapsulates the operation it will perform and then attempts to acquire the resource. For example, the call **putnext(q, mp)** attempts to acquire the resource

associated with **q->q_next**. Before doing so, it places the call, **(*q->q_next->q_qinfo->qi_putp)( )**, along with the two arguments, **q->q_next** and **mp**, into a callback request.

If the resource is free, the caller acquires it and performs the operation as normal. In the example, the next queue's put procedure would be called. This process can be repeated indefinitely until the operation reaches the end of the stream, or encounters a locked resource. In operations on a stream with appropriate synchronization, control will pass successfully from driver to module to stream head without delay.

If the caller finds the resource already in use, for example, by another thread, it enqueues its request on the resource's synchronization queue and simply returns. When the holder of the resource is ready to release it, the OSF/1 STREAMS framework checks for pending requests and executes any it finds. The **putnext( )** call in the example would be performed by whichever thread holds the target queue's resource.

Whether the resource is free or in use, the original call returns promptly. The check for pending requests ensures they execute as soon as the resource becomes available. Because the synchronization queue is FIFO, they execute in the order received.

If the call is to a module's or driver's open or close procedure (that is, in response to **open( )**, **close( )**, **ioctl(I_PUSH)**, or **ioctl(I_POP)**), a special flag is set in the callback request. If the holder of a resource encounters this flag when processing requests, it takes a special path and hands off the resource to the special caller. In most cases, this caller is executing in the stream head, and context then switches to the appropriate user thread.

There are five major advantages to this scheme:

- The synchronization mechanism is completely invisible to the module or driver code.
- Normal callers of STREAMS routines never block, and the calls are never delayed more than the processing time requires. This is the primary means of achieving execution throughput.
- It takes full advantage of multiprocessors; that is, threads that do not contend for STREAMS resources execute fully in parallel. Resources may be independently configured according to the architecture of the STREAMS module or driver.
- No additional context switching is imposed on STREAMS. It is not necessary to provide service procedures to implement the scheme, nor

are context switches performed when requests are processed or called back, except where necessary (in open and close procedures).

- The mechanism resolves a problem of recursion and small kernel stacks. It is very dangerous to call procedures recursively in kernel context with small kernel stacks, and the modularity of the STREAMS specification makes recursion possible. OSF/1 STREAMS will return promptly when it encounters the first recursive acquisition, protecting the stack. However, the action is taken as the resource is released, so the effect is the same as if it were acted on immediately.

13.5.1 Synchronization Queue Structures

As previously mentioned, synchronization queues are implemented as lists of callbacks associated with the resource (queue) that the original caller was trying to access. These synchronization queues are made up of two data structures:

- The *synchronization queue head* structure (SQH), which represents the resource
- The *synchronization queue element* structure (SQ), which contains the callback to the procedure with two arguments, usually a queue's address and a pointer to the message being passed

The SQH is identified in the list of SQs by a bit in a common flag word. Other bits indicate the type of request, for example, whether the request requires execution in context. The lock protecting addition and removal from the queue is located uniquely in the SQH, along with any ownership information.

13.5.2 Changes to Standard STREAMS Structures

OSF/1 STREAMS uses the STREAMS data structures **queue_t** for queues and **mblk_t** for messages as documented in the *AT&T Programmer's Guide: STREAMS*. However, it extends the "framework-visible" portion of the queue structure to support its special synchronization and scheduling mechanisms. These extensions are invisible to the module and driver code. The next two subsections describe these changes.

13.5.2.1 The Queue Structure

The read and write queues for the stream head and driver are allocated when the stream is first set up. For a pushed module, they are allocated when the module is added to the stream.

The standard STREAMS queue contains pointers and data fields describing the procedures that can run on a queue, the messages to be processed, and flow control. The OSF/1 STREAMS version of the structure contains additional fields for use by its special queue synchronization and scheduling functions. The new fields include:

q_sqh A synchronization queue head for synchronizing accesses to the queue. This is the queue's default resource.

q_runq_sq A synchronization queue element used when scheduling the queue's service procedure.

q_act_next A pointer that is used with the **q_act_prev** and **q_thread** fields to form a registry of acquired resources for use when a thread sleeps in open or close procedures.

q_ffcp The forward flow control pointer to the next queue in the stream to be enabled when queue flow control is cleared. The **q_bfcp** field is the backward flow control pointer.

13.5.2.2 The Message Structure

OSF/1 STREAMS messages consist of one or more linked *message blocks*, each of which is a triplet. A triplet consists of the **mblk_t** and **dblk_t** control structures and a data buffer. The data buffer, which is of variable size, holds the actual contents of the message.

Currently in OSF/1, **mblk_t** and **dblk_t** are allocated in contiguous memory, with a small data buffer rounding out the memory block. Messages with larger data buffers have the data buffers allocated separately.

In addition to the standard components of **mblk_t**, the OSF/1 STREAMS version of the structure contains an additional field for use by its special queue synchronization and scheduling functions. The new field is

b_sq A synchronization queue element used for passing the message to a queue's put procedure. This SQ is linked onto the queue's **q_sqh**.

13.5.3 Executing the Synchronization Queue

There are two ways of accessing a resource that is protected by a synchronization queue. (Synchronization queues are described in Section 13.5.)

The first way to access a resource is to pass a callback to the synchronization queue. The callback will be executed immediately if the resource is free, or enqueued if it is not; the caller will not know the difference. The routine that passes a callback is **csq_lateral**( ). **csq_lateral**( ) never blocks, and is used by all stream-level operations that potentially acquire new resources, such as **putnext**( ), **qreply**( ), and so on.

The second way to access a resource is to reliably acquire it with a call to **csq_acquire**( ). As with **csq_lateral**( ), an SQ is prepared, and if the resource is not free, the SQ is enqueued on the synchronization queue. The difference is that there is no callback function, and if the SQ is enqueued, the thread calling **csq_acquire**( ) blocks, waiting for its SQ to be discovered on the synchronization queue. Although **csq_acquire**( ) cannot be called by service procedures within a stream, it is available to stream head routines, which are essentially synchronous in operation.

Unlike **csq_acquire**( ), which functions as a blocking lock on STREAMS resources, **csq_lateral**( ) never blocks but employs a callback mechanism to ensure that its callers execute under appropriate protection. In order to simplify stream head operation, **csq_acquire**( ) is allowed to recursively acquire resources. However, **csq_lateral**( ) never recursively acquires them.

The possible contents of a synchronization queue consist of SQ elements, each of which represents one of the following:

- An anonymous job, that is, a callback enqueued by **csq_lateral()** that is not associated with any thread context.
- A job placed by calls to **csq_acquire()**, with a thread waiting for the resource. These SQs set the special flag **SQ_HOLD**.

Figure 13-2 shows an example of how synchronization queues actually get executed. For simplicity, the example shows busy and simultaneous activity with only a few threads contending heavily for a few resources. An actual situation would have more diverse activity and fewer miraculous coincidences.

In the figure, there are four threads contending for the resource associated with a read queue. Thread 4 is calling **putnext(q, mp)** on the queue from below, while at the same time Threads 1 and 2 are attempting to access the queue through the stream head. Thread 3, the queue's put procedure, has placed the message on the queue, resulting in the service procedure being scheduled.

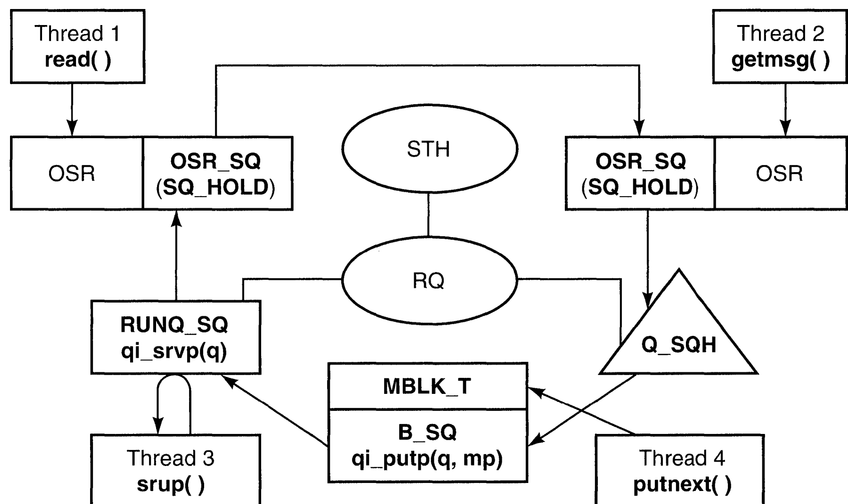
Thread 4, calling **putnext(q, mp)**, is the first to access the resource. It enters **csq_lateral()**, which is granted access and proceeds to call the queue's put procedure. The put procedure (Thread 3) decides to put the message on the queue, scheduling the queue's service procedure. The event is acted on promptly (the machine is a multiprocessor), but when the **netisr** thread uses **csq_lateral()** to invoke **(*q–>q_qinfo–>qi_srvp)(q)**, it encounters the lock Thread 4 still owns on the resource. As a result, the **netisr** thread enqueues the service procedure as a callback and returns. The user threads, 1 and 2, simultaneously perform operations to fetch data from the queue, a **read()** and a **getmsg()**. (Although this situation is highly unlikely, it is useful for illustrating the concepts described in this example.) They allocate OSRs and proceed in parallel to attempt to acquire the queue with **csq_acquire()**. Because the resource is in use, each uses an SQ in its OSR to enqueue itself with **SQ_HOLD**, and both threads block.

Next, the put procedure completes and returns to **csq_lateral()**. **csq_lateral()** finds callbacks on the synchronization queue and proceeds to a routine called **csq_turnover()** to handle them. First on the queue is the service procedure callback, which it handles itself, because **SQ_HOLD** is false in that SQ. Note that no additional context switch occurred when the service procedure was invoked, only the initial one. The service procedure does its job and returns.

At this point, **csq_turnover( )** still sees callbacks pending, but the first one has **SQ_HOLD**. It knows that this represents a thread that is waiting to acquire the resource. Therefore, it hands over ownership of the resource to the waiting thread, issues a **wakeup( )** call to it, and returns. Thread 4 is finished with the resource, and is no longer of interest.

The waiting thread, which is performing a **read( )**, begins executing as the owner of the resource. It removes some of the data from the queue and returns it to the user. When it is done, it releases the resource. There is still another callback pending, so **csq_turnover( )** repeats the turnover sequence with the thread performing **getmsg( )**, and when that completes, the synchronization queue becomes empty. Note that the minimum amount of context switching has occurred to completely process all the requests.

Figure 13–2. An Example of Synchronization Queue Execution



13.5.4 Acquisition of Multiple Resources

STREAMS modules and drivers normally need to acquire only one resource in order to proceed, for example, a queue. Certain stream head operations, however, require that more than one queue or stream head be held. An example is the **I_PUSH** operation, where both the stream head and the currently uppermost module in the stream must be acquired, since pointers in both have to be altered to point to the new module.

In order to prevent deadlocks from occurring between threads that need to acquire the same resources, a global synchronization queue head called **mult_sqh** is statically allocated in the stream head. It is not associated with any particular resource, but it must be acquired first by any thread that intends to acquire more than one resource. Once all the resources are successfully acquired, **mult_sqh** can be released as appropriate.

This rule ensures that deadlocks will not occur between threads proceeding in parallel as they acquire multiple resources. By following the **mult_sqh**-first rule, they will synchronize before attempting any acquisitions that would deadlock.

A difficult consequence of this resource acquisition scheme is that service and put procedures in drivers and modules cannot directly acquire multiple resources. OSF/1 STREAMS provides a solution with the *welding* mechanism discussed in Section 13.8.

13.5.5 Synchronization with Interrupts

The **csq_acquire**( ) routine handles the acquisition of resources from thread context, where it is possible to block, but interrupts require a different solution. Although **csq_lateral**( ) is capable of handling interrupt synchronization, it would require significant overhead to protect each utility routine, such as **putq**( ), **getq**( ), and other queue-handling functions, for interrupt access. Also, it would be impossible to implement **canput**( ) and **qenable**( ) with this restriction. Therefore, two special modifications to the framework have been made to provide interrupt safety while retaining minimal overhead.

The first modification is a queue lock. A **q_qlock** that protects **canput**( ) and **qenable**( ) is added to each queue. This lock also protects low-level queue flow control decisions, such as those performed by **putq**( ), **getq**( ),

and other queue utility functions. This lock protects only the **q_flag**, the **qb_flag**, and counts. The rest of the queue, including the queued messages themselves, is protected by holding the queue's resource.

The second modification is to **putq**( ). In the normal case, **putq**( ), **getq**( ), and other queue routines are called when the queue has already been acquired through its synchronization queue. However, **putq**( ) can also be called by a driver routine operating asynchronously and therefore outside of the usual stream context. This is the case in a STREAMS driver when an interrupt occurs, for example, a terminal driver when a character is received or a network driver when a packet arrives. In this case, execution enters the driver without first being protected by the synchronization queue mechanism. The *AT&T Programmer's Guide: STREAMS* makes clear that **putq**( ) must be used in such a case, and that the driver must provide a read service procedure to handle the message from STREAMS context. (This also minimizes time spent at the hardware interrupt level.)

The **putq**( ) modification checks whether the queue that **putq**( ) is targeting is at the bottom of a stream. If it is, **putq**( ) calls **csq_lateral**( ) in order to first acquire the proper STREAMS synchronization. If the resource is in use (for example, by the service procedure in response to a previous interrupt), the situation works as for any other resource contention—the **putq**( ) is deferred by queuing it as a callback for the current holder of the resource.

To determine whether it is targeting the bottom of a stream, **putq**( ) inspects the queue's flow control pointers. At all other points in the stream, **putq**( ) behaves normally.

The other class of routine that can inspect and set queue parameters from interrupt context includes **canput**( ) and **qenable**( ). These routines perform their functions entirely through the **q_flag**s (or **qb_flag**s in the case of **bcanput**( )), the queue counts, and high and low water marks. Therefore, it is important that a module or driver never attempt to change the value of **q_flag**, except as supported by STREAMS calls such as **noenable**( ) and **enableok**( ), nor such things as queue high and low water marks except by **strqset**( ).

13.5.6 Synchronization of sleep( ) Calls

STREAMS modules and drivers are permitted to sleep in their open and close procedures. This presents a problem to the OSF/1 STREAMS synchronization mechanism because the threads holding resources must not sleep without first releasing their resources.

Before the open or close procedure of a module or driver begins, the identity of the resource holder and the resources it holds are placed into a registry called **active_queues**. The resources are the **mult_sqh**, two or more queues associated with the stream head, and either the module being pushed or popped, or the driver being opened or closed.

Calls to **sleep( )** (as well as **tsleep( )** and **mpsleep( )**) are intercepted from OSF/1 STREAMS by the routine **streams_mpsleep( )**. Currently, this is accomplished at compile time, by redirecting these calls using the C preprocessor, in the **sys/stream.h** header file.

The **streams_mpsleep( )** routine determines the identity of its caller by its thread, and looks up any resources the caller holds. Before the caller can begin sleeping these resources are released, and after the sleeping is completed they are reacquired. Note that when the caller reacquires resources, if the **mult_sqh** was held, it must be reacquired first.

This sequence can cause a timing problem. Because of the rule that the synchronization queue must be executed (that is, drained) before it releases the resource, it is possible in some cases to cause the very event that is being awaited.

Consider, for example, a thread in which an open procedure is waiting for an acknowledgement from a device. If such an acknowledgement is passed in a message generated at interrupt time, the message may arrive while the thread is still executing the open procedure and has not yet released the resource. Therefore, the interrupt will fail to immediately **putq( )** the message, and instead place it on the synchronization queue which the thread currently owns. Because the thread has not yet seen the message (it is unprocessed on the synchronization queue), the thread proceeds to sleep.

However, before sleeping, the thread releases the resource associated with its read queue, and in so doing it calls its own **putq( )**. This delivers the message it intended to wait for. OSF/1 STREAMS avoids the subsequent race by performing an **assert_wait( )** before releasing any resources.

13.5.7 Synchronization of timeout( ) and bufcall( )

The **timeout( )** and **bufcall( )** utilities present a particular problem to the synchronization mechanism. Good programming practice would have the module or driver call only an interrupt-safe routine as the callback. However, historically *all* of STREAMS has been considered interrupt-safe, and such routines have performed no more synchronization than possibly raising interrupt level with **splstr( )**.

Ideally, the only routines passed as callbacks would be fully interrupt-safe, such as **qenable( )** or **wakeup( )**. To support code in which they are not, the solution is an optional configuration for these callbacks from the OSF/1 STREAMS framework.

If the **STR_QSAFETY** bit is set when the module or driver is configured, the same **active_queues** registry is invoked each time a resource is acquired. When **timeout( )** or **bufcall( )** is invoked, this registry is checked for the currently held resource, and the resource is remembered so it may be acquired before the callback is performed. This method imposes an overhead on the module, so it is optional in order to improve performance for those modules that do not require it.

As in the case of **sleep( )**, **timeout( )** calls are redirected with the C preprocessor in the **sys/stream.h** header file. There is no such redirection required for **bufcall( )** because STREAMS implements it directly.

13.6 Memory Allocation

OSF/1 STREAMS performs all allocation through the standard kernel **malloc( )** function. This means all memory is shared among kernel subsystems and leads to the most efficient use of memory. Occasionally, however, memory allocation will fail, especially when it must be constrained by executing in interrupt context. When **allocb( )** fails, STREAMS provides the **bufcall( )** utility to recover.

13.6.1 The bufcall( ) Routine

In OSF/1 STREAMS, the bufcall event is handled in a pseudomodule. The bufcall module has a single queue and is invoked by **qenable( )** when memory becomes available, or at regular intervals in order to poll for memory released by other subsystems. The module cannot be pushed; its implementation is only for modularity with the rest of STREAMS. However, this illustrates the concept that STREAMS provides an architecture that is useful in areas other than device handling.

When a module or driver calls **bufcall( )**, a request structure is removed from a freelist and assigned a unique ID. If no structures are available, **bufcall( )** returns 0 (zero), and many callers will issue a **timeout( )** call to retry later. In the normal case, this request is queued, and a trigger is set for enabling the bufcall module when memory becomes available.

When this wakeup occurs, the bufcall read service procedure makes sure that the memory size in the request is available, and then invokes the callback, also stored in the request. If the **STR_QSAFETY** option was set for the active queue that invoked **bufcall( )**, the presence of the queue in the **active_queues** registry would have also placed it in the request structure. The same queue's resource would simply be acquired before the callback.

13.6.2 Interaction with mbufs

STREAMS data blocks (**dblk_t**s) already support the referencing of external data, as do sockets memory buffers (**mbufs**, described in Chapter 12). For subsystems, such as XTI, which transfer data between the two, OSF/1 provides a mechanism for translating buffers without copying the data. Two routines, **mbuf_to_mblk( )** and **mblk_to_mbuf( )**, are provided for this operation. Detailed calling information can be found in the *OSF/1 System Programmer's Reference Volume 2*.

The STREAMS **dblk_t** may reference a routine to be called when the **dblk_t** is freed. This is normally specified when the **mblk_t** is allocated through **esballoc( )**, as described in the *AT&T Programmer's Guide: STREAMS*. However, calling this routine from the context of **freeb( )** may be problematical in a multiprocessor environment, since that context may be an interrupt. It would be impossible to provide a routine that needed to take certain locks.

To avoid placing such a restriction on routines to free these buffers, OSF/1 STREAMS does the same thing as is provided for OSF/1 mbufs: calls to any free routines of **dblk_t**s are deferred and invoked in thread context. In fact, both mbuf and mblk free events are performed at the same time. Chapter 12 has further information.

13.7 Cloning

There is a special type of **open()** call called *cloning*, which is often used by STREAMS devices. Cloning is a mechanism for obtaining unique invocations of a single name. In the case of STREAMS, where all open streams are character special files, these invocations are created by dynamic allocation and management of *minor numbers*.

It is important for certain classes of STREAMS devices to be clonable, in order to simplify and protect their creation. These devices are normally those that are not associated with hardware, such as pipes, protocol endpoints, pseudoterminals, and so on. For such devices, it is never desirable to reopen them by name without the current owner being willing to allow it. Normally the rights to access such devices are passed to children through open file descriptors across **fork()** or by explicitly allowing access through **fattach()** or **mknod()**.

In order to manage these devices without a complicated (and potentially very large) namespace and protection facility, OSF/1 uses a simple and powerful concept called the *clone device*.

The clone device is configured into the **cdevsw** and consists only of a single procedure, open. Since it is configured into the **cdevsw**, it receives a unique major number, but the minor number is used to represent the major number of another device, such as a STREAMS device.

When the open procedure of the clone device is invoked, it simply passes the open call, along with a special flag indicating a clone open, to the device owning the major number associated with the clone's minor number. All further **cdevsw** operations are vectored to this other device. For example, if the clone device resides at major 24 and the STREAMS echo device resides at major 30, the clonable device node for the echo device would have (24, 30) as its device number. Opens of device (30, x) would specifically open the minor number x of the echo device, while opens of device (24, 30) would have an unused echo minor number dynamically assigned.

A significant amount of work occurs in the vnode and special file handling layers during a clone open in OSF/1, but only very little work needs to occur in the open procedure of a clonable OSF/1 STREAMS device. The routine **streams_open_comm( )** and an associated close routine make the driver's job trivial. The **streams_open_comm( )** routine manages the minor number space for any STREAMS driver, in addition to allocating optional device buffers and handling multiprocessor locking. This routine is a STREAMS-specific wrapper for the underlying **cdevsw_open_comm( )** routines. See the *OSF/1 System Programmer's Reference Volume 2* for more information.

13.8 Welding

In certain circumstances, it is desirable to be able to alter the connections between the queues in a stream. An example is a loopback driver whose write-side queue would connect directly to its own read-side queue. Irregular queue connections of this kind are called *welds* in OSF/1 STREAMS.

Stream connections cannot be directly manipulated by module routines in a multiprocessing environment. This is because more than one queue resource must be held by the thread that wants to connect or disconnect them, and the OSF/1 STREAMS synchronization mechanism requires that a thread acquire multiple resources before this can occur. Since this is only possible in the stream head, STREAMS modules and drivers cannot manipulate their queue pointers directly.

OSF/1 STREAMS uses an asynchronous request mechanism to weld and unweld queues. Within modules and drivers, the **weldq( )** routine can be called to weld, and the **unweldq( )** routine can be called to unweld, up to two pairs of module or driver queues. These routines prepare a request that is passed as a **netisr** request to the weld routines, which acquire the necessary resources and perform the **q_next** and other pointer manipulations.

The arguments consist of two or four queue pointers to be pairwise welded to one another, and a routine and argument to be called upon completion. Because the actual welding or unwelding is done outside of the context of the caller, **weldq( )** and **unweldq( )** must asynchronously notify the

requestor. The specified callback function is generally **wakeup()** or **qenable()**, and the requesting thread can detect completion by inspecting the passed **q_next**s.

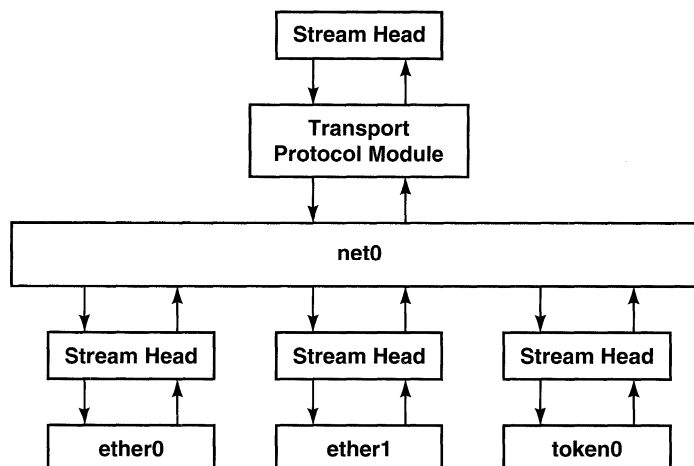
The **weldq()** and **unweldq()** routines are documented in the *OSF/1 System Programmer's Reference Volume 2.*

13.9 Multiplexing

STREAMS allows multiple streams to be connected beneath the driver of a special multiplexing stream. A STREAMS multiplexer can associate multiple streams with a single controlling stream; for example, a system's network devices can be linked together as lower streams beneath a master "network stream," as in Figure 13-3. The multiplexing device shown as **net0** is a logical router implementing an OSI level 3 network protocol.

It is up to this driver to handle the routing of data into and out of the correct lower streams. The driver indicates its ability to be a multiplexer by setting **st_muxrinit** and **st_muxwinit** in its **q_info** structure.

Figure 13–3. Lower Streams Multiplexed to a Master Stream



An application manages this new stream configuration by **ioctl( ) I_LINK** and **I_UNLINK** (or **I_PLINK** and **I_PUNLINK**) calls. Almost all the work is done in the stream head.

13.9.1 Multiplexing Lower Streams

When a stream is to be connected under a multiplexing driver, special put and service procedures are substituted into the lower stream head by replacing its **q_info** fields with the contents of the driver's multiplexer **qinit**s. Data is also copied from the **module_info** pointed to by the multiplexer **qinit**s.

However, the multiplexing driver at the bottom of the stream must perform other operations. It will probably have to initialize some internal structures, and will likely reset the lower stream head's **q_ptr** to point to them. At the least, it will have to be notified that another lower stream now exists and must accept or reject the **I_LINK** or **I_PLINK** request. Finally, the synchronization levels for the newly linked stream head must be adjusted as necessary to match the needs of the multiplexer driver. The lower stream head is in effect taken over by the driver.

The system-level stream head first performs some basic checking on the ability of the driver to serve as a multiplexer and on the status of the two streams. If these are acceptable, it proceeds by building the **M_IOCTL** message that will later be passed to the multiplexing driver. The **linkblk** in this message contains the information needed to make the link, as specified in the *AT&T Programmer's Guide: STREAMS*.

The system-level stream head then acquires (by using **csq_acquire( )**) the lower stream head and also the read queue below it to prevent messages coming up while the linking is active. Various values are stored in the lower stream head, such as flags indicating the stream is linked, and a unique multiplex ID is selected. The lower stream head put and service procedures are reset to the multiplexing driver's. Finally, the lower stream head's synchronization level and synchronization queue are reset to those of the driver.

At this point, the upper stream head's work is done. All that remains is to let the driver know what has happened. The **M_IOCTL** message is passed to the driver's write put procedure.

Here, synchronization again takes effect. The upper stream head must effectively call **sleep( )** to wait for the reply from the multiplexing driver indicating its willingness to accept the newly linked stream. Before this reply (a message passed to **sth_rput( )**) can be received, six resources must be released—the upper stream head's two queues, the driver's two queues, and the lower stream head's two queues.

If the acknowledgement message is received and indicates success, the operation is done; all resources will be released and properly set. If the message times out or indicates an error, the entire process is reversed and the multiplexer reverts to two separate streams.

13.9.2 Unlinking Multiplexed Lower Streams

Unlinking lower streams from a multiplexer is the reverse of linking them, except for one special circumstance: multiplexed lower streams are not unlinked when they are closed; they remain active. When they are unlinked from their upper streams, the unlink processing must be able to recognize them and complete their close.

In the case that the lower stream head is still open, the unlink operation must be careful to restore the various fields it reset in the link operation to

their former values. Unlinking the lower stream head does not require saving its linked values. The unlink operation merely restores the nominal stream head values.

13.10 Initialization and Configuration

The OSF/1 STREAMS framework is initialized and configured at system startup time. The initialization process:

- Initializes internal STREAMS data structures
- Initializes the memory allocation and **bufcall**( ) mechanisms
- Initializes the **weld**( ) function
- Registers the **netisr**s for the scheduling, timeout, and weld mechanisms
- Configures all the statically-bound STREAMS modules and drivers

The STREAMS-based tty and pty subsystems, XTI interface, and other modules and drivers can also be dynamically configured into a running system. To add them, the OSF/1 system administrator issues the **sysconfig** command to have the **cfgmgr** daemon install them.

13.10.1 Driver and Module Configuration Options

OSF/1 STREAMS provides compatibility options for source code that is written to older versions of the STREAMS specification, or that requires additional synchronization from the framework. Refer to the description of **strmod_add**( ) in the *OSF/1 System Programmer's Reference Volume 2* for more information.

The configuration options are

STR_QSAFETY
: Supplies STREAMS synchronization for **timeout**( ) and **bufcall**( ) callbacks, as described in Section 13.5.7.

STR_SYSV4_OPEN
Calls the module's or device's open and close procedures using the System Version 4 calling sequence. If this bit is not specified, the System Version 3.2 calling sequence is used. See the *AT&T Programmer's Guide: STREAMS* for the appropriate specifications.

13.10.2 Synchronization Levels

The OSF/1 STREAMS synchronization mechanism offers flexible selection of synchronization levels. The choice of a synchronization level is made for each STREAMS driver and module based upon the structure of its code and data. In general, it is chosen to minimize the scope of the resource and to maximize STREAMS execution throughput by avoiding contention.

At configuration, each module or driver exports its STREAMS data structures and constants to the framework in the **streamadm** passed to **strmod_add**( ).

Valid synchronization levels are

SQLVL_GLOBAL
The resource will be a global synchronization queue. All modules under this lock are thus single-threaded. Note there may be modules using other levels not under the same protection. This option is available primarily for debugging.

SQLVL_MODULE
Module-level synchronization. All code within this module or driver will be single-threaded. An example is a module that maintains shared state, such as a TCP module's port binding registry.

SQLVL_ELSEWHERE
The module is synchronized with some other module. This level is used for synchronizing a group of modules that access each other's data. A name passed along with this option is used to associate with other modules; the name is decided by convention among cooperating modules. For example, a networking stack, such as a TCP module and an IP module, both of which share data, may agree to pass the string **tcp/ip**.

SQLVL_QUEUEPAIR
Queue-pair synchronization. Code executing on the read or write side of this queue will be single-threaded. Other queues in this module or driver may execute in parallel. This is a common synchronization level for most modules that process data and have only per-stream state, such as a TTY line discipline.

SQLVL_QUEUE
Single-queue synchronization. The read and write sides of this queue may execute in parallel. This is the lowest level of synchronization available from the OSF/1 STREAMS framework. It is used by modules with no need for synchronization, because either they share no state, or provide their own synchronization or locking.

Because the stream head acts as a loopback for **M_FLUSH** messages, and in order to simplify the coding of the stream head, all stream head queues are synchronized at the queue-pair level.

The mechanism for implementing these synchronization levels is quite simple. Each queue contains, in addition to a synchronization queue header, a pointer to a synchronization queue header. This pointer indicates the actual SQH to use to synchronize each queue's resource:

- For **SQLVL_QUEUE**, it points to the queue's own SQH.
- For **SQLVL_QUEUEPAIR**, both queues point to the read queue's SQH.
- For **SQLVL_MODULE**, both sides of all queues belonging to the module or driver point to a per-module or per-driver SQH.
- For **SQLVL_ELSEWHERE**, both sides of all queues belonging to the module or driver point to a per-module or per-driver SQH, except that the SQH is dynamically allocated when the identifier string is first encountered.
- For **SQLVL_GLOBAL**, both sides of all such queues point to a single global SQH.

SQHs can be recursively acquired, so the stream head and other routines that acquire multiple resources can simply proceed successfully after initially acquiring each shared resource.

13.11 Streams Security Extensions

OSF/1 STREAMS can be configured with the following security extensions:

- The architecture and internal interfaces have extensions that associate security attributes with each message that traverses a stream, and that define the attribute format used at the interface between the stream head and its downstream neighbor.
- The OSF/1 STREAMS programming interface has extensions that allow programs to obtain the security attributes associated with received messages, that allow trusted applications to specify the attributes to be attached to the messages they send, and that define the attribute format used at the programming interface.
- Hooks can be configured for auditing data transfers that result from **ioctl**( ) calls, such as **getmsg**( ), which is an implemented **ioctl**( ) call, and special STREAMS **ioctl**( ) calls, such as **I_PEEK** and **I_FDINSERT**.

Compatibility has been maintained with existing programs, modules, and drivers that use the STREAMS programming interface. Chapter 15 describes the security extensions.

The only change to STREAMS data structures in a secure configuration is to the **mblk_t** structure, which receives a single new field:

b_attr A pointer to another **mblk_t** containing the security attributes of the originator of the message. The attributes are shared with any **b_cont** **mblk_t**s linked to this message.

Handling of attributes of each message is normally performed only at the stream head, and in OSF/1 Release 1.1 the attributes are not visible to the module or driver writer.

Chapter 14

OSF/1 Logical Volume Manager

In traditional UNIX systems, each file system must completely reside on a single *physical volume* (a disk drive or portion of a disk drive). The Logical Volume Manager (LVM) subsystem provides a level of abstraction between physical volumes and the file management subsystem that allows a file system, or even a single file, to span multiple physical volumes.

14.1 Overview

The LVM implements and manages *logical volumes*, each of which can represent one or more physical volumes. From the kernel's perspective, a logical volume looks like a physical volume, and the LVM looks like the device driver that manages the logical volumes.

The LVM provides the following features for managing disk storage in OSF/1:

Disk spanning
: The LVM enables file systems and raw partitions to span multiple physical disks, without requiring modifications to existing system and application software. The amount of data stored in a single file system can exceed the amount of disk space actually available on any one disk in the system.

Dynamically resizable volumes
: Unlike a physical disk system, whose volume sizes cannot be changed, the logical volume storage area can be resized dynamically. This allows an administrator to expand the amount of disk space allocated to a file system or partition without the time-consuming process of backing up the data, reinitializing the file systems, and restoring the data.

Replication
: Replicated, or mirrored, data provides data reliability in the event of hardware failure, such as when a disk sector becomes defective. It can also enhance the performance of some applications by maintaining multiple copies of a block for faster access.

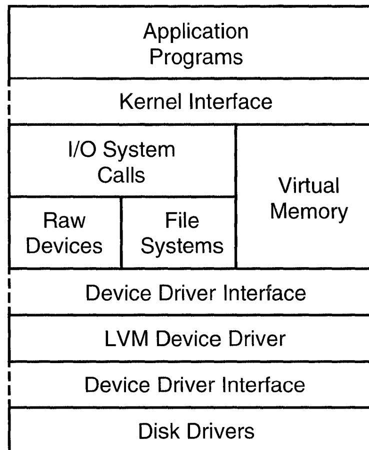
In addition, the LVM also provides the following:

Bad sector relocation
: The LVM can detect and revector bad sectors that develop on a disk drive. If the disk driver supports hardware bad block replacement, the LVM will automatically use it; otherwise, the LVM remaps the defective sector in software.

Dynamic configuration capability
: The LVM subsystem can be dynamically configured into the OSF/1 kernel at runtime, rather than statically configured at system build.

The main component of the LVM subsystem, the LVM device driver, creates and exports logical volumes. The LVM device driver maps the storage space for the logical volumes onto the physical volumes. Figure 14-1 illustrates the relationship between the LVM driver and the rest of the operating system.

Figure 14–1. Relationship of the LVM to Other System Components



This chapter assumes that the reader is familiar with the UNIX I/O system, traditional UNIX disk subsystems, and UNIX device drivers.

14.2 LVM Terms and Concepts

This section describes some terms and concepts that will help in understanding the LVM. It includes:

- A list of LVM component terms
- A description of LVM mirroring
- A description of LVM quorums
- A description of logical-to-physical mapping

14.2.1 LVM Component Terms

The following list describes general LVM component terms. These terms refer to disk drives and their logical and physical components as the LVM sees them.

volume — A block storage device. It corresponds to a disk drive, or a disk partition in a traditional UNIX system. It is also used to refer to a logical volume implemented by the LVM.

physical volume — A contiguous area of a physical disk drive. This can mean either an entire disk or a portion of the disk (for example, a UNIX partition). Physical volumes are specified to the LVM through device pathnames, such as **/dev/dk0c**. Usually the LVM uses an entire physical disk as a physical volume, but it allows for using individual UNIX partitions as physical volumes.

logical volume — A volume implemented by the LVM. To users and file systems, logical volumes appear as devices. There are block and character device nodes to perform I/O and system commands on the logical volume. A logical volume can be thought of as a virtual disk drive, although it may map to multiple physical volumes.

volume group — A set of physical and logical volumes, and the mappings between them. Logical volumes can only map to physical volumes that are in the same volume group. All of the administrative and error recovery features of the LVM center around the volume group.

physical extent — The unit of allocation of physical volume space. All of the physical extents within a given volume group are the same size. The LVM restricts the extent size to be a power of 2 between 1 MB and 256 MB.

logical extent — Each logical volume consists of a number of logical extents. These logical extents may, but do not need to, map to physical extents. Logical

extent and physical extent sizes are equal within a volume group. Because the logical and physical extent sizes are equal, this chapter often uses the generic term *extent size* to indicate both sizes.

Each logical extent can map to 0, 1, 2, or 3 physical extents. When a logical extent maps to 2 or more physical extents, the extent is mirrored. A logical extent that is mapped to no physical extent cannot be used to store data.

disk sector
: The smallest unit of I/O to a physical disk, and hence to a physical volume. This is typically 512 bytes. The sector size is significant to the LVM in that it is the unit of defect relocation. The extent size must be a multiple of the sector size. The term *block* is often used as a synonym for sector.

page size
: The smallest unit that is typically read or written by the OSF/1 system. This size is significant to the LVM because multiple I/O operations within the same page are serialized. Serialization is necessary for the LVM's fault recovery mechanisms to work. It is possible to perform I/O on less than a page size area. The operation is suspended until any other I/O to the same page finishes.

The page size is typically either 4KB or 8KB. It is chosen by the system vendor to be the smaller of the VM page size (see Chapter 6) and the file system block size.

logical track group
: Each logical track group (LTG) consists of 32 consecutive pages. The LVM uses the LTG to ensure mirror consistency, and to perform mirror resynchronization. The size of the LTG cannot be larger than the maximum physical disk I/O length. The LTG is not visible outside of the LVM driver.

14.2.2 Mirroring

The LVM can improve data reliability by replicating the data that is stored in a logical volume. Data can be *singly mirrored* (one additional copy) or *doubly mirrored* (two additional copies). If the data is singly mirrored, two identical physical extents (containing the replicated data) are assigned for each logical extent. If the data is doubly mirrored, three identical physical extents are assigned for each logical extent.

The LVM can transparently recover from the loss of one copy of the data by retrieving another (mirrored) copy of that lost data. Depending on how many mirrored copies of a block of data there are, the LVM can redirect I/O intended for the missing data to a secondary or tertiary copy of the data.

The consistency of the mirrored copies is maintained by the LVM, so that the application will always be guaranteed to read the same data, regardless of which mirror copy the data is from.

14.2.3 Quorums

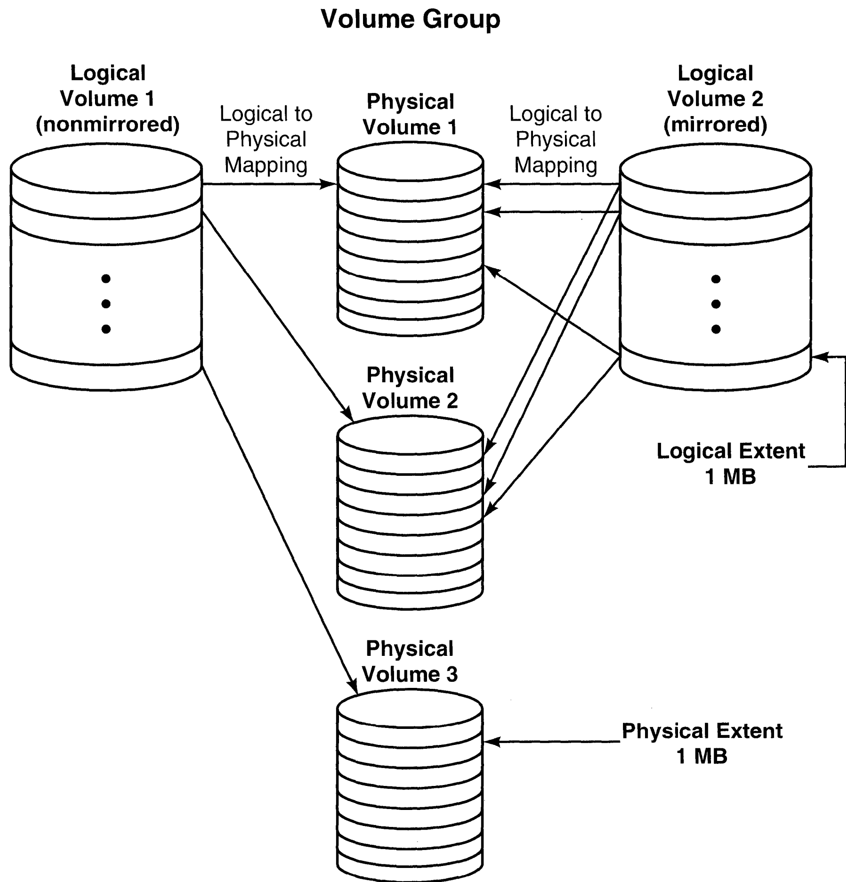
The LVM uses the concept of a quorum to keep track of volume group state. A quorum is the majority of a group of physical volumes (more than half) that must contain identical descriptor information. This guarantees that the LVM is operating with accurate physical volume state information (such as the volume group ID, the number of currently installed physical volumes that are in the volume group, and so on). Without a physical volume quorum, any operations that update the volume group state or configuration information are disallowed.

14.2.4 Logical-to-Physical Mapping

Applications see the logical volume as a normal block device. The LVM driver maps the logical volume requests to physical volume accesses to store and retrieve the user's data. This translation is completely transparent to the application. Each physical and logical volume is sectioned into physical and logical extents, respectively. Figure 14-2 illustrates the relationship between logical and physical volumes. As the figure shows, separate device drivers interface to the logical and physical volumes. In the

example, one of the logical volumes has been mirrored, or replicated, onto two physical volumes.

Figure 14–2. A Mapping of Logical to Physical Volumes



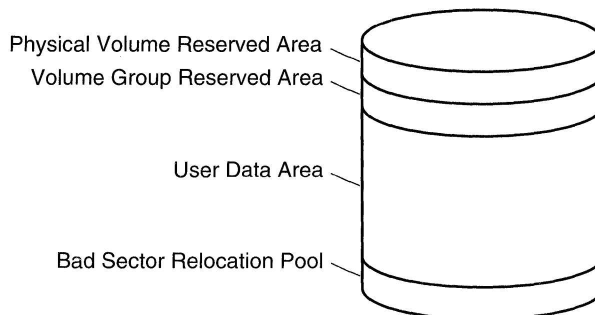
14.3 LVM Disk Layout

Because logical volumes are mapped to physical volumes for the storage of data, this section describes how data is arranged on the physical volumes. The LVM driver stores configuration and information data on reserved portions of the physical volumes. The physical volume is separated into four regions to store the information. These areas include:

- The physical volume reserved area
- The volume group reserved area
- The user data area
- The bad sector relocation pool

Figure 14-3 shows the general layout of a physical volume.

Figure 14–3. Physical Volume Layout



14.3.1 Physical Volume Reserved Area

The physical volume reserved area (PVRA) contains the structures describing the configuration of the disk drive. This information consists of the LVM record and the bad sector directory. Every physical volume managed by the LVM contains a PVRA.

The LVM record contains the following information:

- The physical volume unique identifier
- The unique identifier of the volume group the physical volume is a member of (if it belongs to a volume group)
- The amount of space available on the physical volume
- The current size of physical extents on the physical volume
- The space allocated for each physical extent, in sectors

The physical extent space must be at least as large as the physical extent size. The physical extent space can be made greater than the physical extent size to provide additional storage for bad sector relocation.

The bad sector directory records all of the sectors that have been software relocated (where a new sector from the bad sector relocation pool is assigned for the bad sector), as well as all of the sectors that have been diagnosed as bad and are waiting for relocation.

Duplicate copies of both the LVM record and the bad sector directory are also maintained in the PVRA for reliability.

14.3.2 Volume Group Reserved Area

The volume group reserved area (VGRA) describes the volume group that contains this physical volume. It consists of three data areas:

- The volume group descriptor area (VGDA)
- The volume group status area (VGSA)
- The mirror consistency record

The VGDA and VGSA are maintained on physical volumes. Their presence on a physical volume is optional, and is specified when the physical volume is installed into the specified volume group. The physical volumes containing these areas participate in the quorum calculation (see Section 14.2.3). If a physical volume has a VGDA and VGSA, the LVM maintains two copies of both of these areas. Two copies of the mirror consistency record are also maintained on all physical volumes in the volume group. The following subsections describe these data areas.

14.3.2.1 Volume Group Descriptor Area

The VGDA is used at the activation of the volume group. It contains configuration information such as the volume group ID number, the maximum number of logical volumes allowed in the volume group, the number of currently installed physical volumes in the group, and lists and descriptions of logical and physical volumes in the volume group. This region is maintained under control of the administrative commands.

For the LVM to decide that it has a valid VGDA, it must locate a quorum of physical volumes in the group. This guarantees that the LVM always operates with the most recent configuration information.

14.3.2.2 Volume Group Status Area

The VGSA describes and maintains the current state of physical volumes in the volume group. It contains one bit of state for each physical volume in a group, plus one bit of state for each physical extent in each physical volume. Other fields in this area contain the maximum number of physical volumes allowed in the volume group and the maximum number of physical extents on any physical volume. The information in the VGSA area needs to be constantly updated to ensure data integrity. For example, if an operation affects the state of a physical extent, the associated bit in the VGSA must be updated to reflect this changed state. The LVM driver updates the VGSA automatically to reflect the volume group state.

14.3.2.3 Mirror Consistency Record

The mirror consistency record contains entries for all regions in the volume group that may be inconsistent due to write operations in progress. The mirror consistency record is always written to one of the physical volumes containing the physical extents that are the mirrors of the logical extent being modified. Following a system crash, only the regions in the volume group marked in the most recent mirror consistency record need to be resynchronized.

If a physical volume is offline when the volume group is activated, the LVM driver can reduce the number of extents it must resynchronize in the following way. Since only the most recent mirror consistency record is used

for recovery, the driver assumes that any physical volume that is offline when the group is activated, but was online when the group was last active, might contain the newest mirror consistency record. Only those logical volumes that are mirrored onto the missing physical volume are assumed to be out of synchronization, and are forced to be resynchronized. The number of required resynchronizations needs to be reduced only if a physical volume goes offline at the same time the system crashes. If the VGSA can be updated to note that the physical volume is offline, then the driver can tell that it could not contain the most recent mirror consistency record.

14.3.3 User Data Area

The user data area stores user data, which may be a file system, virtual memory paging space, or application data. The user data area is divided into fixed-size extents (physical extents), and all allocation of physical disk space to logical volumes is performed in units of these extents.

14.3.4 Bad Sector Relocation Pool

The bad sector relocation pool region consists of disk sectors to which the LVM device driver redirects the I/O intended for defective disk sectors. The LVM device driver handles both soft and hard disk errors.

If an uncorrectable error occurs on a read operation, the LVM will first redirect the I/O to a mirror copy (if one exists) to obtain the data, then allocate a sector from the bad sector pool and write data into the new sector. This restores full replication of the mirrored data.

14.4 Programming Interfaces

The LVM has two programming interfaces. The User Application Programming Interface is used by applications that use an LVM logical volume to store data. The Administrative Application Programming Interface is used by system administration applications that manage the configuration and operation of the volume group and the volumes it contains.

14.4.1 User Application Programming Interface

The LVM uses the general UNIX system calls that allow applications to open, manipulate, and close files:

open( ) Opens a logical volume

close( ) Closes a logical volume

read( ) Reads from a logical volume

write( ) Writes to a logical volume

ioctl( ) Performs control operations on a logical volume

The **LVM_OPTIONGET** and **LVM_OPTIONSET** operations allow the application to exercise control over raw (character) device operations.

For information about these system calls, see the *OSF/1 Programmer's Reference*.

14.4.2 Administrative Application Programming Interface

The LVM uses the **ioctl( )** function to control the configuration and operation of the volume group and the logical and physical volumes that it contains. This interface is used by the administrative commands to carry out the operations requested by the system administrator.

For information about the **ioctl( )** interfaces used by the administrative commands, refer to the **lvm** reference page in the *OSF/1 Programmer's Reference*.

14.5 LVM Device Driver Architecture

This section describes the data structures and external entry points into the LVM device driver, and provides a high-level description of the driver operations.

All parts of the LVM device driver are parallelized for multiprocessor operation. Locks are used around critical data structure accesses to ensure that information is not corrupted.

14.5.1 Data Structures

The LVM device driver uses both internal and on-disk data structures (those on the VGRA) to describe the volume groups, logical volumes, and physical volumes that it is managing. The LVM device driver uses internal data structures to process I/O requests and keep track of the state of physical and logical volumes. The main structures are described in this section.

The LVM device driver uses the UNIX block buffer structure, **buf**, to keep track of logical requests, and uses an LVM-specific structure, **pbuf**, to keep track of physical requests. The **pbuf** is a **buf** structure with additional fields to keep track of the correspondence between physical requests and logical requests. It includes information for bad sector defects and the physical volume that the request is intended for, and also maintains a list of requests. There is at least one **pbuf** structure associated with each **buf** structure being processed by the LVM.

The volume group structure, **volgrp**, is allocated when the volume group is configured into the system. This structure contains general information about the group, such as the extent size, number of logical and physical volumes, number of open logical volumes, and pointers to an array of logical volume and physical volume structures.

The logical volume structure, **lvol**, is allocated when the volume group is activated. It contains information about a specific logical volume, including a work-in-progress hash table, the logical and physical extent arrays, scheduling policy functions, and logical volume option flags.

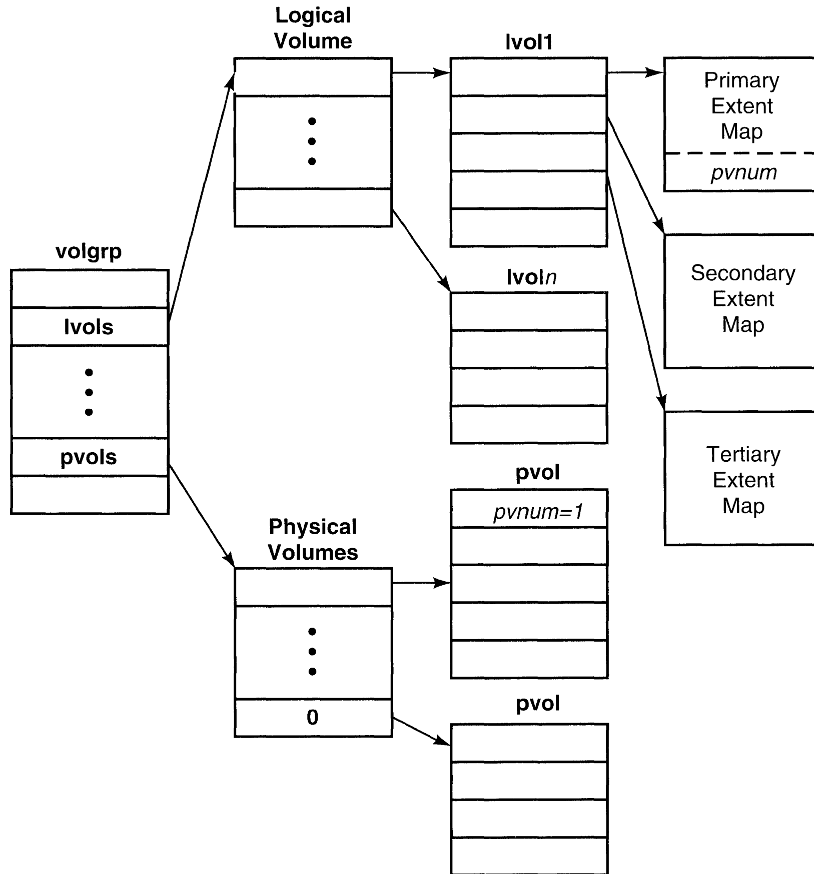
The **extent** map structure is allocated when the volume group is activated. This map contains one entry for each logical extent in the logical volume. If

the logical extent is mapped to a physical extent, then each entry contains the physical extent number and the physical volume number that the extent resides on.

The physical volume structure, **pvol**, is allocated when the associated physical volume is attached to the volume group. It contains information about a specific physical volume, including pointers to structures for the bad sector directory on the physical volume and the volume group structure that the physical volume belongs to.

Figure 14-4 illustrates the relationships between the volume group data structures. The primary, secondary, and tertiary blocks refer to the mirrored copies of the extent maps.

Figure 14–4. Data Structures Describing a Volume Group



14.5.2 Driver Entry Points

The LVM device driver is a normal device driver with character driver entry points for **open**, **close**, **read**, **write**, and **ioctl**, and block device entry points for **open**, **close**, and **strategy**. The LVM device driver calls into the **strategy** entry point in the physical disk driver to perform actual I/O operations.

lv_open( ) Called when a logical volume is mounted as a file system, or the device is opened for block or character access.

lv_close() Called when a logical volume is unmounted or when the last **close()** occurs on the open file corresponding to the device.

lv_read() Called by the **read()** routine to translate the character I/O requests to block I/O requests.

lv_write() Called by the **write()** routine to translate the character I/O requests to block I/O requests.

lv_ioctl() Called by the **ioctl()** routine to manage the volume group. The **lv_ioctl()** routine performs the specified request. (Refer to the **lvm** reference page in the *OSF/1 Programmer's Reference* for a description of the possible requests.)

lv_strategy()
Provided for block device requests. It processes logical block requests, and takes care of overlapping requests for these requests.

To the OSF/1 kernel, the LVM device driver behaves like an ordinary disk driver.

14.5.3 Flow of Control

For a raw I/O request to a character logical volume device, the device driver works as follows:

1. An application issues a **read()** or **write()** on a file descriptor corresponding to the character special file for the logical volume.
2. The LVM device driver **lv_read()** or **lv_write()** entry point is called through the character device switch to process the read or write request.
3. The LVM device driver I/O routine calls the kernel **physio()** routine, which performs much of the raw I/O work and issues one or more requests to the **lv_strategy()** entry point, as necessary.
4. The **lv_strategy()** entry point receives the I/O request. The LVM driver performs all of the mapping, mirroring, bad sector relocation, and any resulting status updates required by the request. For each physical I/O operation required, the LVM driver issues a request to the physical volume driver.

5. The physical volume driver **strategy** routine arranges to transfer the data. On completion, the physical volume driver calls the kernel **biodone**( ) routine, using the physical request buffer header. The **biodone**( ) routine then invokes a callback in the LVM driver.

6. Once the driver determines that the logical request is complete, it invokes the **biodone**( ) routine with the logical request buffer header to notify the original requestor that the I/O is complete.

File system and block device I/O requests to the LVM device driver are passed directly to **lv_strategy**( ) through the block device switch, where they are processed in the same manner as raw I/O requests (starting from step 4).

14.6 Driver Theory of Operation

A high-level description of the LVM device driver flow of control for a raw I/O request was provided earlier in the chapter. This section provides a more detailed description of the operations the LVM device driver performs.

The LVM device driver is divided into the following layers:

- Configuration and raw I/O layer
- Strategy layer
- Mirror consistency manager layer
- Scheduler layer
- Status area manager layer
- Physical layer

An LVM request is processed by each layer before being passed to the next layer. Once the I/O request is complete, it returns through each layer to the requestor. Some of the layers initiate their own I/O requests that must be completed before a request from a higher layer can proceed. For example, a mirrored write must be blocked at the mirror consistency layer until the mirror consistency record has been written to the disk.

14.7 LVM Configuration and I/O Layer

The LVM configuration and raw I/O portion of the driver implements the programming interface to the LVM driver. This includes both configuration operations and raw device I/O. This portion of the driver maintains the operational information required by the remainder of the driver, and initiates I/O operations necessary to change the configuration.

14.7.1 Driver Dynamic Configuration

The LVM device driver can be dynamically configured into the OSF/1 kernel at runtime, rather than statically configured at system build. When the OSF/1 system receives a configuration command for the LVM, it invokes the LVM configuration routine to allocate and set up volume group structures. It also registers the LVM device driver entry points in the device switch table. When the LVM receives a valid unconfiguration request, it deallocates all of the allocated resources. The **sysconfig** command controls the performance of these operations.

14.7.2 Volume Group Configuration

The administrative application programming interfaces can be divided into two classes: those that modify the volume group configuration, and those that return information about the current state of the volume group. The interfaces that return information do not cause any disk I/O transfers to occur because the LVM device driver maintains in-memory copies of all configuration and status data structures. The requested information is looked up, formatted according to the interface definition, and returned to the caller. The interfaces that change configuration information are more complex because they need to write any changes to the physical volumes, and also need to ensure that operations in progress are not disrupted.

To simplify the driver operations, the LVM driver maintains two versions of the configuration and status information. One version is an exact image of the information as it appears on the physical volume, in the PVRA, VGDA, or VGSA. The second version uses the internal data structures described in Section 14.5.1. The internal version controls the actual operation of the volume group, while the on-disk image represents the state after any

configuration change that is in progress completes. The order of modification of the two sets of structures depends on the type of operation to be performed; sometimes the internal version is updated before the on-disk version, sometimes afterwards. These two versions are described in the following subsections.

14.7.2.1 Modifying the On-Disk Data Structures

The LVM driver creates a special logical volume, called the control device, to maintain the VGRA. The VGRAs from all physical volumes in the volume group are mapped into this control device. This way, the data structures can be read or updated through an appropriate I/O request to the **lv_strategy**( ) routine.

Interfaces that update the VGRA must write multiple copies of the area so that if the system crashes, the change will be visible after reboot. A change is only considered permanent if it is propagated to a quorum of physical volumes.

The operation of updating these multiple locations can be time consuming (and may fail), so the LVM driver does not suspend other operations to wait for the VGRA updates to complete.

14.7.2.2 Modifying the Internal Data Structures

The internal data structures used by the LVM to describe the configuration of a logical volume are not allowed to change while a logical volume I/O operation is in progress. This simplifies the bad block and mirror consistency management code, as they need not be concerned with changes in extent allocation between operation initiation and completion.

The configuration layer is able to change the configuration information while a logical volume is in use by *pausing* the logical volume. This causes the strategy layer (described in Section 14.8) to block any future I/O requests. Once any in-progress I/O requests complete, the configuration layer makes the necessary modifications to the data structures, and then *continues* the logical volume. The configuration layer does not perform any I/O operations between the pause and continue, so the logical volume operation is not disrupted significantly.

14.7.2.3 Example: Deallocating A Mirror from a Logical Volume

As an example of a configuration operation that could result in loss of data if not implemented correctly, the deallocation of a mirror from a mirrored logical volume is described in this section.

In this example, the configuration layer must stage the modifications to the internal and on-disk structures to guarantee that user data will never be lost as the result of a system crash in the middle of a configuration operation. The order in which modifications are performed is important.

The deallocation (**LVM_REDUCELV**) of physical extents from a mirrored logical volume proceeds as follows:

1. The input parameters are checked for validity, to minimize the errors that can occur during the operation.
2. The logical volume is paused. This guarantees that there are no mirror writes in progress.
3. The in-memory image of the on-disk VGDA is modified so that it does not include the deallocated extents.
4. The internal extent maps are marked so that the mirror consistency manager will not use the to-be-deallocated mirror as a location for the mirror consistency record. (See Section 14.3.2.3 for more information about the mirror consistency record). The mirror is still present in the extent map, so it will continue to be accessed normally.
5. The logical volume is continued.

   At this point, none of the on-disk data structures have been changed.
6. The in-memory image of the VGDA is propagated to the physical volumes. Once the first VGDA is modified, the change will be permanent if that VGDA is present on reactivation. Once a quorum of physical volumes has been updated, the change is guaranteed to be permanent.

   While this operation is in progress, the internal extent maps still show the old allocation. This ensures that if a crash occurs before the configuration change is committed, any user data will be correctly written to all mirrors.

   Once the new VGDA is committed to disk, the LVM driver will recognize that the mirror was deallocated when the volume group is reactivated, even if the system crashes.

7. The logical volume is paused again.
8. The internal extent maps are updated to reflect the new configuration, without the deallocated mirror.
9. The logical volume is continued, and the configuration change is complete.

This procedure ensures that data is updated correctly. If the structures were updated in a different order than described here, there would be a risk that the mirror would not be deleted from the VGDA description, and that data would not be up-to-date. If the system crashed while this state existed, user data would be corrupted.

14.7.3 Raw I/O Layer

The raw I/O layer is entered whenever the LVM device driver receives a character I/O request. The read or write entry point transforms the request into a block request by first generating a raw buffer for the request, then calling **physio( )**.

The **lv_strategy( )** routine, which processes the block request.

14.8 Strategy Layer

The strategy layer processes logical block requests from the configuration and I/O layer, the VM subsystem, and the block buffer cache. It validates and serializes logical block requests before passing them to the next layer (mirror consistency management).

When the strategy layer receives an I/O request, it first ensures that the request is in whole disk blocks, and that it does not cross logical track group boundaries. The strategy layer rejects requests that it judges invalid, and notifies the requestor through a call to the **biodone( )** routine.

If logical requests overlap block ranges, the strategy layer serializes these requests in First-In-First-Out (FIFO) order. Serializing requests at this layer ensures that bad block relocation and mirror synchronization are performed correctly in the lower layers. The strategy layer uses a work-in-progress queue to keep track of all outstanding requests. If an overlapping request

arrives, the strategy layer blocks that request until all earlier requests to the overlapping pages complete. When each request finishes, requests that are blocked waiting for it are then processed. Logical requests to overlapping block ranges complete in FIFO order.

The strategy layer also controls access to the logical volume configuration data structures. By controlling these accesses, the strategy layer ensures that the data structures remain static for the lower layers of the driver while an I/O request is in progress. When the configuration layer needs to modify the configuration of a logical volume, it first pauses the logical volume (blocking new I/O operations), and waits for all in-progress operations to complete. The configuration data structures are updated, and then the logical volume is released. Section 14.7.2.2 describes this process.

The strategy layer also initializes the logical volume options for the request. These options include bad block relocation, write verification, and mirror write consistency.

Once it has verified that the request is valid and does not conflict with earlier requests, the strategy layer sends it to the mirror write consistency management layer for further processing.

14.9 Mirror Consistency Management Layer

The mirror consistency management layer manages the operations required to maintain mirror consistency. The mirrors must be consistently maintained so that an application always receives the same data regardless of which mirror copy this data was obtained from. The LVM mirror consistency manager guarantees that two consecutive reads of the same logical block with no intervening write always have the same result, even across a system crash. This condition must be maintained to ensure that applications do not see unexpected results.

The data on a mirror can become stale if the LVM driver cannot update all copies of the data when the mirrored logical volume is written. This can occur if the system crashes before all mirrors have been updated, or if one or more mirrors are not available when the write occurs. The LVM monitors these conditions, and prevents reads of stale data from the out-of-date mirrors. When the data becomes inconsistent, mirrored copies of data need to be resynchronized to restore the replication. The mirror consistency

management layer recovers data from an accessible copy of the data and writes this data to the remaining mirrors to reestablish mirror consistency.

The mirror consistency management layer passes all requests that do not need mirror consistency checking directly to the scheduler layer. This includes all reads, any nonmirrored writes, and any write for which the mirror write consistency is not required.

If mirror write consistency is required, the affected area must be marked as "in transition" in the mirror consistency record before the write is allowed to begin. If the mirror consistency manager needs to update the mirror consistency record to guarantee the described condition, it blocks the request, and initiates an I/O request to write the record to a physical volume. When the consistency record update is complete, the original write request is then passed to the scheduler layer.

The mirror consistency record entry persists until no writes are outstanding to the area, and the entry is needed for a different region of a logical volume. The entry will also be removed if the logical volume is closed. This algorithm attempts to minimize the additional disk writes needed to maintain mirror synchronization across crashes, while placing an upper bound on the amount of mirrored data that may need to be synchronized following a crash.

14.10 Scheduler Layer

The layers preceding the scheduler layer deal with logical I/O requests that refer to a block within a logical volume. The scheduler layer converts these logical I/O requests into physical I/O requests, and then initiates the actual I/O operations.

14.10.1 Scheduling Policies

How the LVM handles a logical request depends on whether the data that it is accessing is mirrored or not, and what type of mirroring is applied. This can be controlled by the scheduling policy selected for the logical volume. There are currently three LVM scheduling policies: the reserved policy, the sequential policy, and the parallel policy.

The reserved policy, which has no mirroring, schedules operations for the volume group control device. This is the simplest of all of the policy scheduler routines because logical volume 0 only contains the volume group descriptor area and volume group status area, and cannot have mirrors. Only one physical buffer needs to be allocated for each logical operation. This policy is not available for use by logical volumes other than the control device.

The sequential policy accesses the primary mirror first, then the secondary mirror, and finally the tertiary mirror. In the case of the read operation, each subsequent mirror is read only if the previous read fails. This requires only one physical buffer to be allocated for the logical operation, regardless of the number of mirrors.

The parallel policy performs writes for all of the mirrors of a logical volume simultaneously. This policy requires one physical buffer for each mirror of the logical volume, up to three buffers. For read operations, it selects the physical volume judged to be the best available. This is the physical volume with the fewest outstanding I/O operations.

The LVM driver is structured so that other scheduling policies may be easily added to the source code.

14.10.2 Scheduler Operations

All logical requests require physical buffers before they can be scheduled. When the scheduler layer receives a request from the mirror consistency layer, it places the request on the scheduler's physical buffer pending queue. Each request is removed from this queue when enough physical buffers become available to initiate the I/O, according to the selected policy.

Once the physical buffers are allocated, the scheduler uses the logical volume configuration information to determine the physical volume and the physical block number that corresponds to the requested logical block. The physical buffers are initialized with this information, and then passed on to the status area manager.

When the status area manager completes a physical request, the scheduler determines if the logical request is complete. For a mirrored read operation, the scheduler will attempt alternate mirrors if the physical request fails. For a mirrored write operation, the scheduler must initiate and wait for all the writes to complete before declaring that the logical operation is complete.

When the scheduler layer is finished with a request, it notifies the mirror write consistency layer.

14.11 Status Area Manager

The status area manager maintains the state information in the VGSA. (Refer to Section 14.3.2.2 for a description of the volume group status area.). When a volume group is activated, this information is read. This information is later updated to account for volume group configuration changes or I/O errors. This area tracks which physical volumes are online, and whether the physical extents contain valid (nonstale) data.

The status area manager inspects each request from the scheduler, determines whether the request should be allowed, and whether the request should result in a status area update. The status area manager disallows reads from stale mirrored data, and causes such requests to be returned to the scheduler without performing any I/O. A write request can cause the status area to need to be updated in several ways:

- The physical layer indicates a write failure.
- The write is not initiated because the physical volume is offline.
- A resynchronization operation completes.

If any of these events occur, the status area manager marks the change in the VGSA, and initiates a write to all copies of the VGSA in the volume group. The request that caused the status update is released once the VGSA has been propagated to all locations.

If write errors make it impossible to update the volume group status area, then the mirror consistency record entry for the failed request cannot be released. This occurs when the volume group has lost quorum, which means that an insufficient number of physical volumes are present to guarantee that the operation is permanent.

14.12 LVM Physical Layer

The physical layer is responsible for initiating and terminating physical I/O requests, and it detects, corrects, and relocates any bad sectors found on the physical volume during operations. The physical layer maintains the bad sector directory on each physical volume.

When the physical layer receives a request, it checks for any known bad blocks. If there are one or more bad blocks, the physical request is split up into multiple requests as follows: each region of the request that contains no bad blocks is processed, and then a separate request is issued to handle the known defect. This is repeated for all defects found in the request. The physical layer processes these separate requests sequentially. When the entire physical operation is completed, the physical layer notifies the status area manager.

14.12.1 Revectoring Known Defects

If a physical request contains a known, relocated defect, the physical layer simply substitutes the physical block location indicated in the defect directory. The physical driver reads or writes the alternate location, and the operation continues.

14.12.2 Detecting New Defects

If the physical driver encounters a bad sector during the I/O, it sets the error fields in the **pbuf** structure to indicate that there is a media error on the disk, and calls **biodone( )** to notify the LVM driver physical layer. The physical layer then performs the required bad sector processing.

If a read operation encounters a bad sector, the physical layer cannot immediately relocate the sector because it does not have the data that the sector should contain. In this case, the physical layer puts an entry in the physical volume defect directory. This entry labels the sector as bad, with no relocated sector address, and with a status indicating that sector relocation is desired.

At this point, the physical request is terminated and returned to the scheduler. If the scheduler cannot locate another mirror copy to read the data from, an I/O error will eventually be returned to the original requestor.

14.12.3 Relocating and Repairing Defects

If a physical request is a write and it contains a sector that is waiting for relocation, the request is sent to the physical device driver with hardware relocation requested. This allows the physical volume driver to relocate and repair the defect if it is able.

If the physical volume device driver indicates that it has successfully relocated and rewritten the defective sector, the physical layer can delete the defect directory entry for this sector. From the standpoint of the LVM driver, the defect no longer exists. If the physical volume driver cannot relocate the defect with hardware, then the physical layer of the LVM driver performs software relocation. It assigns a new sector from the bad sector relocation pool (see Section 14.3.4) and substitutes the new location for the bad sector. It then updates the defect directory on the physical volume to indicate that the sector has been successfully relocated.

14.12.4 Dynamic Detection, Relocation, and Repair

The scheduler layer can use the operations described in the preceding sections to dynamically detect, relocate, and repair defects on a mirrored logical volume.

If a mirrored read operation encounters a bad sector, the scheduler layer first performs the operations necessary to read a good copy. The scheduler layer then converts that successful mirror read into a write request, and the physical layer is reinvoked. The physical layer detects a write operation to a known defect that needs to be relocated, and proceeds with the normal relocation operation.

This technique maintains full mirror replication of all blocks in a logical volume, even when defects develop on any of the mirrors.

Chapter 15

Security

The primary goal of the OSF/1 security architecture is to consistently enforce a security policy. A security policy, as defined by the National Computer Security Center (NCSC) in its statement of trusted computer system evaluation criteria (commonly called the Orange Book because of its color), is "a set of rules that are used by the system to determine whether a given subject can be permitted to gain access to a specific object."[1]

This chapter provides an overview of the OSF/1 security features, followed by details of their implementation. Because the security system is motivated by government requirements, this chapter includes an explanation of the NCSC security model and often includes references to the Orange Book.

1. *Trusted Computer System Evaluation Criteria (TCSEC)* (CSC-STD-001-83), U.S. Department of Defense, National Computer Security Center, August 15, 1983. Requirement 1.

15.1 Security Overview

There are two main reasons for making a system secure. One is that system purchasers often demand specific levels of security, certified through formal evaluation. Many government contracts now require stricter levels of security than C2. (See Section 15.2 for information about security levels.) OSF/1 is B1 secure, with some features beyond B1. The other reason is safe performance in the commercial arena. Secure systems offer protection against industrial espionage, crackers, and human errors, which have the potential to misuse or destroy data.

OSF/1 uses a combination of command authorizations, privileges, login user IDs (LUIDs), and auditing to trace operations back to a particular user. In contrast, in traditional UNIX systems, many users can be running as **root** and sharing the root password, and there is no easy way to record which user performs which actions. OSF/1 does not use the **/etc/passwd** file, which in traditional UNIX systems is visible to all users, to store passwords. In OSF/1 they are stored in a protected database, and a variety of authentication mechanisms have been added.

OSF/1 security features revolve around the interaction between *subjects* and *objects*. Subjects take active roles in operations, and include users and processes. Objects have a passive role, and include files, directories, character and block special devices, pipes, symbolic links, message queues, pseudo-ttys, shared memory segments, semaphores, UNIX domain sockets, and processes in certain cases. For example, when processes are targets for signal delivery, they are objects. When a user is running a debugger process to debug another process, the process being debugged is an object.

The OSF/1 security features include the following:

Access control mechanisms

OSF/1 uses the following mechanisms to control access to objects:

- Access control lists (ACLs) for objects allow owners to specify who can have what access to their data. This extends the normal UNIX discretionary access control (DAC) policy by increasing its flexibility. DAC attributes can be changed at the discretion of the owner of the object.
- Mandatory access control (MAC) involves enforced restrictions on objects, which cannot be changed at the

discretion of the creator or owner or user. Subjects are assigned levels of trust (clearances), and objects are assigned degrees of sensitivity. Both notions are represented with sensitivity labels and combine a hierarchical classification with a nonhierarchical set of categories (or compartments).

Authorizations and privileges

To reduce the need for users (and thereby processes) to run as **root**, OSF/1 uses authorizations and privileges. Command authorizations restrict certain operations to designated users. Kernel authorizations grant certain security policy overrides to trusted applications. Privileges give processes rights to access operating system functions. Each process has a kernel authorization set, which is the set of privileges for which the process's user is authorized.

A process also has a base privilege set and an effective privilege set. The base privilege set is the set of privileges that is always granted to a process when it executes a file. The effective privilege set is the set of privileges that the process is actually using when the kernel checks for privileges.

Two privilege sets are associated with each executable file. The potential privilege set is the set of privileges allowed for anyone who runs a program, but which are not enabled until the program makes a specific request. The granted privilege set is the set of privileges allowed and enabled for any process that runs a given program. Granted and potential privileges are assigned to the binary copy of a program. The granted privilege set is a subset of the potential privilege set, and allows privilege manipulations in cases where the source for a program is not available (and the program was written with no privilege consideration, typically programs that would be **setuid** to **root** on traditional UNIX systems).

Used in combination, privileges and command authorizations allow suitably authorized ordinary users to do routine privileged tasks without having to run as **root**. For example, with the **sysadmin** command authorization, a user could routinely reboot the system, run **fsck** on file systems, repartition disks, and do most other system administration tasks without having to be logged in as **root**. This is both

more secure (since commands like **rm** do not generally run with privilege when the user is not logged in as **root**) and more convenient.

Least privilege and privilege bracketing
OSF/1 employs the principles of using the fewest privileges (least privilege) for the shortest time (privilege bracketing). Trusted applications and library routines run with as few rights as necessary to accomplish their tasks. Routines are supplied for enabling and disabling privileges, so that these privileges can be enabled for only the time during which they are used.

15.2 The Orange Book Model

The Orange Book defines criteria for classifying computer systems according to their degree of protection. These classifications are as follows:

Division D Contains those systems whose security features have been evaluated and have failed to meet the requirements of any higher division.

Division C A very minimal level of security containing two classes. To be in class C1, a system must provide controls so that users can protect private information and keep others from *accidentally* reading or destroying data. The model is of cooperating users processing data at the same levels of security.

To be in class C2, a system must meet all of the requirements for class C1, and, in addition, users of the system must be individually accountable for their actions through login procedures, auditing, and resource isolation. The recommended method for resource isolation is ACLs.

Division B To be in class B1, a system must meet all the requirements for class C2 and must, in addition, have an informal statement of the security model, and must provide data labeling and mandatory access control over named subjects and objects. The capability must exist for accurately labeling exported information (for instance, in a defense environment, Top Secret printouts must be clearly labeled as such).

To be in class B2, a system must meet all the requirements for class B1 but, instead of an informal statement of the security policy model, there must be a clearly defined and documented *formal* security policy model. The discretionary and mandatory access controls of B1 must extend to *all* subjects and objects, much more thorough testing and review is required, and stringent configuration management controls are necessary.

To be in class B3, a system must meet all requirements for class B2; in addition, its trusted computing base (TCB); that is, that portion of the system that runs in privileged mode, must be small enough to be subjected to rigorous analysis and test. All accesses of subjects to objects must be mediated, the system must be tamper-proof, a *security administrator* must be supported, audit mechanisms must be expanded to *signal* security-relevant events, and detailed system recovery procedures must be in place.

Division A The primary difference between class A1 and class B3 is that the formally specified design must be formally verified.

Certification requires a formal evaluation process. It is not merely the operating system that is being evaluated, but also the implementation of the operating system on a particular architecture with a given set of options. Most of the voluminous documentation required for certification at B1 is supplied with OSF/1. These criteria, currently, are strictly for standalone systems. A system that is networked cannot be secure under these criteria. For example, a B1-certified system cannot contain NFS.

15.3 Security Extensions

OSF/1 can be configured with a variety of security features, the sum of which enables the system to achieve a security level consistent with the B1 level, as defined by the NCSC. The system can also be configured to be consistent with the C2 level, or with other combinations, regardless of

NCSC criteria. These features, and the conditionals allowing them to be configured, are as follows:

SEC_BASE Auditing, process privileges, kernel authorizations, identification and authentication enhancements, other miscellaneous enhancements[2]

SEC_PRIV File-based privilege sets

SEC_ACL_POSIX
Access control lists (ACLs) (based on the POSIX P1003.6 draft 11)

SEC_MAC Mandatory access control (MAC) (access depends on a comparison of the subject's clearance with the object's sensitivity level)

These conditionals are called *base* conditionals. Source that is common to more than one base conditional is placed under a *derived* conditional. For example, source common to both ACLs and MAC is placed under **SEC_ARCH**. Source common to both these and file-based privileges is placed under **SEC_FSCHANGE**. These derived conditionals are defined in **sys/secdefines.h**, and can be extended to include additional security policies.

The security modifications to the OSF/1 base can be summarized as follows:

- New system calls extend the security capabilities of the kernel.
- New library routines provide services to application programmers to make use of the added security features of the kernel and other trusted application support features.
- Added user-level commands provide users with a set of tools for querying the system about security attributes, and for manipulating the security attributes of subjects and objects.
- Security-specific data structures added to the kernel support new security routines.

2. For example, if configured with **SEC_BASE**, OSF/1 does not create a core file when a privileged program is abnormally terminated because that might allow unwanted access to privileged data.

- Modified data structures store the additional security information needed by the new security routines.
- Hooks placed in existing system calls and user-level commands at security-relevant points call the new routines in the security library or the security-specific system calls, and use the data structures that support security operations.

For a complete description of the OSF/1 security design, source licensees can read the *OSF/1 Security Detailed Design Specification*.

15.4 The Trusted Computing Base

The trusted computing base (TCB) is defined by the Orange Book as "The totality of protection mechanisms within a computer system—including hardware, firmware, and software—the combination of which is responsible for enforcing a security policy. A TCB consists of one or more components that together enforce a unified security policy over a product or system. The ability of a TCB to correctly enforce a security policy depends solely on the mechanisms within the TCB and on the correct input by system administrative personnel of parameters (for example, a user's clearance) related to the security policy."[3]

The TCB mediates all accesses between TCB-implemented subjects and objects. The act of associating a subject with its attributes is the task of the identification and authentication (I&A) component of the TCB. All subjects make accesses to objects, each of which is labeled with identity-based protection attributes and, for B1 level systems, a sensitivity level. The access mediation components of the TCB enforce access control in the following ways:

- Between the subject identity and the object discretionary protection attributes (DAC)
- Between the subject and object sensitivity levels (MAC)

3. *Trusted Computer System Evaluation Criteria (TCSEC)* (CSC-STD-001-83), U.S. Department of Defense, National Computer Security Center, August 15, 1983, p. 116.

- Between a subject's sensitivity level (as defined by the security policy model) and the clearance of the user with which that subject is associated, which must dominate the subject's sensitivity level[4]

The TCB also provides a mechanism for analyzing all security-relevant actions in the system by implementing an audit subsystem, which records event records to an audit trail.

The security policy model is enforced on all software objects and mechanisms implemented by the system's TCB. The TCB must maintain proper relationships between subjects and objects for all of the operations defined for the various software objects, as well as provide the necessary mechanisms to implement functional requirements stated by documents such as the Orange Book.

The TCB is organized into two major functional units separated by a well-defined interface:

- The body of code that executes in the hardware's privileged mode—the operating system or kernel
- The body of code that executes in execution domains (processes) running without the hardware's privileged mode—the nonkernel TCB

The system call interface separates these two components.

Processes that implement some aspect of the system's security policy make up the nonkernel TCB. The trusted programs that are included in the nonkernel TCB are organized into subsystems, each of which provides some service (for example, authentication or printed output) to other (nontrusted) processes.

Each subsystem may include programs that are invoked automatically by the system, typically at system startup (daemon programs) and programs that users invoke to perform services (helper programs). Trusted programs are normally distinguished by nonempty privilege sets, indicating that the process executed from that program has access to kernel services reserved for trusted applications. For a list of trusted files, see the *OSF/1 Security Features Administrator's Guide*.

---

4. Sensitivity levels are compared based on a dominance relationship. Level **A** is said to dominate level **B** if **A**'s classification is greater than or equal to **B**'s (according to numeric value of the classification) and if **A**'s compartments are a superset of **B**'s. Dominance is discussed in Section 15.9.

The unsecured OSF/1 system that the system is based on provides some basic services that must be considered trusted. The secured OSF/1 system modifies these services, and adds many of its own to implement the full set of nonkernel TCB programs.

The nonkernel TCB components include:

System initialization and shutdown

The kernel hand-crafts the **init()** process during system initialization, and transfers control to the entry point of that program. The **init()** process is the first code to run without hardware privilege on the system and is the process ancestor of every process subsequently created. System shutdown occurs through cooperation of the **init()** process with a set of system termination programs that gracefully transition the system to a quiescent state. The file system buffer cache is flushed (through **sync** operations), and some processes (such as database managers) are given time to do their own cleanup before the rest of the system shuts down.

Security databases

A major function of the nonkernel TCB is to authenticate users to the system so that an accountable identity can be established for that user's processes. The security databases are accessed and manipulated by the system's TCB to set and enforce parameters associated with users, import/export devices, and system files.

File system maintenance

The on-disk format of file systems, when OSF/1 is configured with **SEC_FSCHANGE**, is that of an extended format file system. An extended format file system includes additional fields for each file describing the file's ACL, sensitivity label, privilege characteristics, and other security-relevant parameters, depending on the system configuration. All fields are allocated, even though all may not be used in a given configuration. The set of commands (for instance, **newfs** and **fsck**) that manipulate file systems is considered trusted because the commands initialize and maintain file system partitions that contain the attributes of files enforced by the operating system when the file system is mounted. Most of the file system maintenance commands are versions of unsecured OSF/1 commands that are modified to support both extended

and unsecured format file systems. Several new commands have been introduced to manipulate the additional security attributes in the extended format file system.

Protected subsystems

A protected subsystem consists of a set of programs, data files, and devices that are protected under a specific group identifier, which is not available to normal user processes. These subsystems implement a set of services that are available through the program interfaces to the subsystem programs. In the trusted system, the mechanisms used to protect data and the mechanisms used to define user authorization within a subsystem (for example, to assume the role of subsystem administrator) have been unified under a common approach and a common set of library interfaces.

Data import/export

A specific requirement of the Orange Book B1 class is the need to appropriately label the data imported to and exported from the system. In the trusted system, this includes printed output and magnetic media. The data import/export subsystem provides line printer and magnetic media software that enforce these requirements. The line printer subsystem labels banner pages of all output with the sensitivity label of the process producing the output and internal pages with the sensitivity label of the file printed. The magnetic media software enforces strict rules for single-level and multilevel tape formats, allowing data to be exchanged between systems with different security policy configurations and different bit representations of labels.

Audit subsystem

The audit subsystem implements the accountability requirements for after-the-fact analysis of security-relevant events that occur on the system. Part of the audit subsystem resides in the operating system itself, while the administrative and reporting parts of the system reside outside the kernel in trusted processes.

Policy support programs/daemons

The system can be configured with a set of security policies that implement access decisions and security attribute maintenance using a combination of kernel and process

components. The kernel component consists of a security policy module, which interacts with the rest of the kernel through a set of routines defined in the security policy switch. The security policy module also interacts with a security policy daemon, which provides mapping services to a database maintained by the daemon. The database layer, implemented through a set of library interfaces, is common across all security policy implementations.

The issue of whether something is part of the TCB is more controversial for libraries. The code in **libc** is not privileged, and cannot directly do anything in violation of the security policy. However, every privileged program shipped with OSF/1 (except for the kernel) depends on **libc**, and could be harmed by changes to **libc**. Therefore, **libc** is indirectly part of the TCB. The same applies to the shells, the **.cshrc** and **.profile** files, and so on.

Although the compiler is not privileged and does not have any code directly related to security policy in it, any defects in it can affect the operation of the privileged parts of the system. Therefore, it, too, is indirectly part of the TCB.

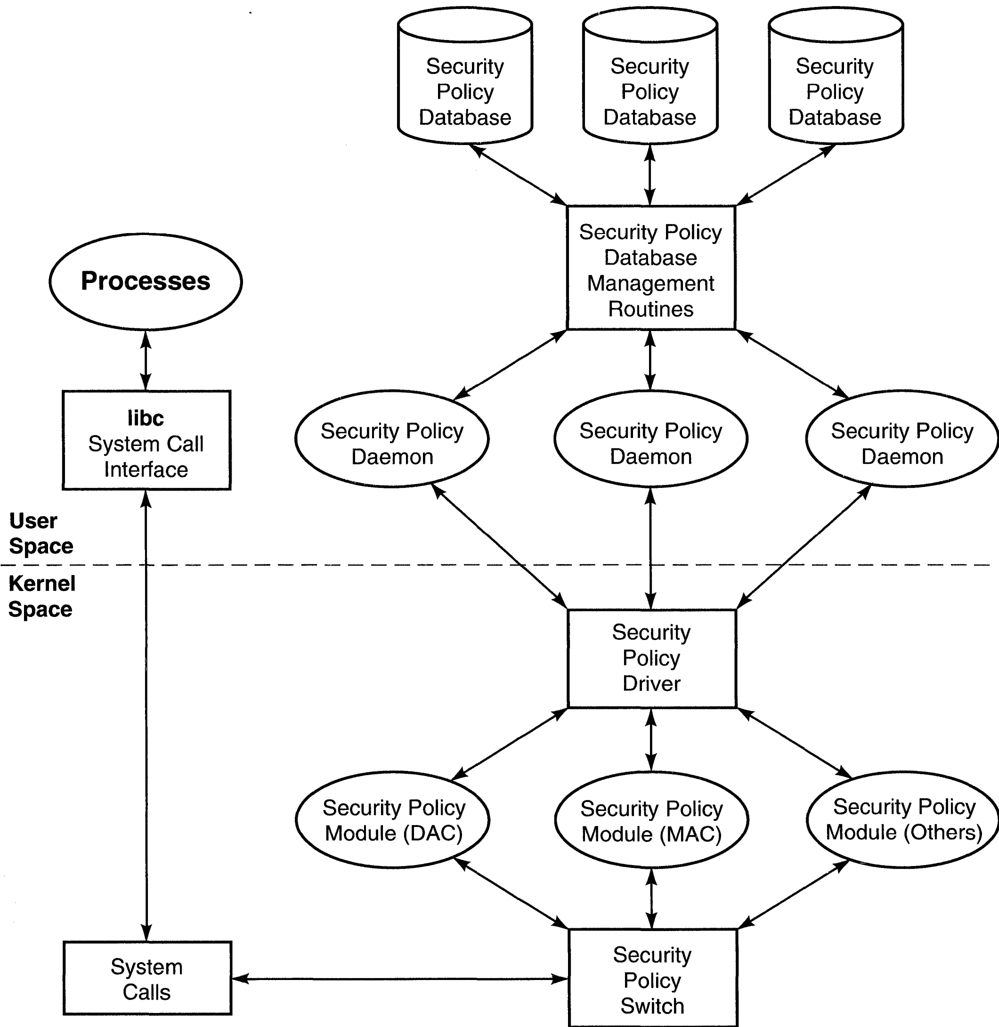
Some data files are part of the TCB. For example, protected password entries are part of the TCB because they can influence the behavior of **login**. If someone was able to alter the **root** password stored there, for example, that person would have free reign over the system.

15.5 Security Policy Architecture

The security policy architecture is the interface between code that wants an access decision made and the code that makes the access decision. Security policies are optional features that may or may not be configured into the system at compile time. Each security policy is implemented through a separate security policy module, which makes access decisions and maintains security attributes relative to that specific policy. The system can be configured with different combinations of security policies, just as it can be configured with different configurations of device drivers. The OSF/1 system, as shipped, knows only two policies, DAC and MAC.

Figure 15-1 illustrates the security policy architecture.

Figure 15–1. The OSF/1 Security Policy Architecture.



The OSF/1 security policy architecture consists of the following major components:

Security policy switch
A table containing information (tag allocation parameters and security policy module entry points) for each security policy configured into the system. (See Section 15.5.1 for a description of tags.)

Security policy modules
(One for each configured policy), modules that contain entry points for each security-relevant operation defined by the architecture. Each module maintains a decision cache of most-recently-made access decisions for performance reasons.

Security policy driver
A software device that passes messages between policy modules and trusted processes (clients) and the policy daemons (servers). The driver does not interpret or operate on the contents of the messages it passes.

Security policy daemons
(One for each configured security policy), daemons that maintain and query the security policy database and are responsible for all access decisions between security attributes relevant to the policy.

Security policy database management routines
Routines that do the following for the security policy database:

- Initialize it
- Close it
- Manage statistics
- Retrieve tags and internal representations[5]

5. Attributes have internal and external representations. The internal representation is binary, and the external representation is human-readable.

- Insert tags and internal representations
- Delete tags and internal representations

Security policy databases
(One for each configured security policy), databases that contain entries mapping tags to and from internal representations.

15.5.1 Security Policy Modules

A security policy module contains entry points for routines that appear in a security policy switch data structure. The security policy configuration of the system is defined by the security policy entries in the security policy switch. The fields in the security policy switch define security-related operations such as object creation, access checking, and security attribute assignment. These security attributes can be complicated; for example, an access control list can be arbitrarily long. A direct representation of such attributes would be too complex for the kernel to manipulate easily. Instead, each attribute is represented by a tag.

Tags are kernel encodings of policy-specific information representing security attributes for a subject or object. A tag pool is associated with each UNIX data structure that describes a subject or object. The tag pool contains the tags that represent the security attributes for that subject or object.

Each security policy module maintains the attributes assigned to it in the subject and object tag pools. The allocation of tags to security policies is also contained in the security policy switch. The entry for each policy in the switch also contains a set of function entry points that call functions contained in the policy module.

The hooks that are implanted in the kernel call the functions defined in the switch for one policy or for all policies when a security-related decision needs to be made or a security attribute-related action needs to be taken. For example, the access functions for all policies are called when a process opens a file for reading, and the change attribute function for the discretionary policy is called when a process tries to change a file's ACL.

15.5.2 Security Policy Daemons

Tags identify a unique security attribute; the meaning of the attribute can be derived only from the data structure that describes the attribute at the system programming interface. The data structure associated with an attribute is called the internal representation, and the format of the internal representation is specific to the policy that maintains the attribute. When a policy module is called upon to make an access decision between two tags, it must arrange for the internal representations associated with those tags to be compared. Associated with each policy configured into the system is a database that contains mappings from tags to internal representations and from internal representations to tags. The database for each policy is maintained by a security policy daemon, which is responsible for the following:

- Allocating new tags when a process specifies an internal representation that is not in the database.
- Making access decisions, when requested by a policy module, by retrieving the internal representations for tags and comparing them. The policy module stores the decision in a cache so that future comparisons between the same two tag values do not require a daemon access. The format of an access decision returned by the daemon and stored in the cache is specific to each policy.
- Looking up the internal representation for a tag when a process queries a security attribute.
- Returning an existing tag when a process requests that a security attribute be set.

The security policy daemon uses a common set of database routines to implement the mappings between tags and internal representations.

The security policy daemon communicates with the kernel through request and response messages. All security policy daemons answer a set of stylized messages that implement the security functions described.

15.5.3 Security Policy Driver

A security policy driver is the mechanism the security policy module and the security policy daemon use to communicate. The driver passes messages between the policy daemons and both the security policy modules and trusted applications. Although there may be many active security policies on a system, there is only a single security policy driver that supports separate software devices for each policy.

The driver is implemented as a pseudodevice, and has the following functions:

- Upcalls (that is, requests sent to the daemon). These requests are implemented by having the daemon make a **read()** system call. The call blocks until the pseudodevice driver has an upcall to make. When the **read()** call returns in the daemon, the result contains the upcall request.
- Buffer management (for holding message requests).
- Process synchronization between the policy daemons and the modules or trusted processes requesting the policy daemon's services.

15.5.4 Security Policy Database Manager

When the system is mediating access requests between subjects and objects, the policy daemon must retrieve the internal representations of the security attributes for the subject and object from its database, unless the decision for the tags comparison between subject and object resides in a cache. It uses the tags of the subject and object as keys into the database for the retrieval.

A security policy database manager manages this database, which maps between the allocated security tags and the internal representation of the security attributes. The database subsystem consists of library subroutines that interface with the database to retrieve, insert, and delete tags and internal representations, and perform other related support functions.

15.5.5 Interactions Example

When the kernel must make an access decision between the process security attributes and the file security attributes (for example, during an **open( )** system call), it retrieves the subject and object tag pools associated with the process and inode, and calls a helper routine through a kernel hook to cycle through the entry points defined in each policy's security policy switch entry. Each policy module checks its decision cache for the decision between the tags.

If the decision is in the cache, the module retrieves the access decision from the cache.

If the decision is not in the cache, the module formats and sends a message with the two tags to be compared to its associated security policy daemon through the security policy message driver. The message driver delivers the message to the policy daemon through the server minor device. The daemon receives the message into its process address space as a result of a **read()** system call to the server minor device associated with the security policy.

The daemon queries its database, retrieving the internal representations associated with the tags and comparing them. The daemon makes all access decisions between the internal representations at this time, because the overhead of an access decision is typically far less costly than the message exchange with the kernel. The daemon formats a decision message and uses a **write()** call to the policy server device to return the decision to the kernel.

Meanwhile, the process thread that initiated the access decision waits for a response (the security policy message driver synchronizes requests and responses). When the response arrives, the policy module loads the decision into the cache and checks the decision for the requested access.

15.6 Privileges and Authorizations

The trusted system does not normally provide any special treatment to processes with effective user ID 0 (zero) or any other process discretionary identity. Rather, the system grants trust based on the authorization attributes of users, and on the privilege attributes of processes and programs. For compatibility, the system offers a mechanism to allow accounts and applications that run with traditional user ID 0 behavior, but this mechanism is implemented by granting the appropriate privilege when the process is operating in a specified mode and its effective user ID is 0.

A process obtains privileges both by inheriting them from its ancestors and through the execution of trusted programs. The mechanism for designating a program as trusted depends on whether the system is configured with file-based privileges (that is, whether or not **SEC_PRIV** is defined). In a system configured with **SEC_PRIV**, specific privileges can be associated with individual executable files, as described later in this section. In a system without **SEC_PRIV**, trusted programs are identified by the presence of set-user-ID (SUID) or set-group-ID (SGID) bits, as in unsecured OSF/1.

Privileges are grouped according to the following categories:

- Those defined to divide the power associated with user ID 0 (zero) in a traditional UNIX system into finer-grained rights that may be individually granted. An example is the **sysattr** privilege, which allows a process to invoke system calls that change system attributes such as the time of day.

- Those defined to restrict the ability to perform a UNIX function that is unrestricted by traditional UNIX systems. An example is the **execsuid** privilege, which allows a process to execute SUID programs.

- Those defined to operate the process in a mode that causes the kernel to treat the process differently than other processes with respect to one of the trusted system features. An example is the **suspendaudit** privilege, which stops the kernel from collecting most system call audit records on behalf of the process.

- Those defined to control access to new privileged functions provided by the trusted system. An example is the **writeaudit** privilege, which allows a process to append records to the audit trail.

- In a system configured with **SEC_PRIV**, two privileges are defined (**sucompat** and **supropagate**) that allow a program to operate in a mode that appears like traditional UNIX treatment of user ID 0 (zero).

Command authorizations control user access to programs or program subfunctions. The system restricts certain activities to certain users by allowing the user to perform the action if he or she possesses the required authorization. Authorizations are enforced by trusted applications and protected subsystems.

Command authorizations are defined and enforced without any kernel support. Some command authorizations are directly related to privileges the kernel enforces, while others control access to services implemented by trusted applications. They control a user's ability to invoke trusted applications or protected subsystems, or to use certain functions of such programs. Most operational rights in the system are allocated to users by the information systems security officer (ISSO), who assigns appropriate command authorizations to accounts on the system (see Section 15.7 for more information on the role of the ISSO).

Command authorizations enforced by commands on a user basis are divided into

- Authorizations that enable the use of a trusted command to perform a specific function. An example is the **mknod** authorization, which allows the invoker of the **mknod** command to create special device files.

- Authorizations that allow a user power to perform tasks associated with one of the system administration roles. An example is the powerful **isso** authorization, which grants the user the ability to administer the security aspects of the system.

- Authorizations that allow additional rights in the programs of one of the system's protected subsystems. An example is the **lp** command authorization, which allows a user the ability to use the administrative commands and command options of the **lp** (line printer) protected subsystem.

A full list of OSF/1 privileges and command authorizations (and their use) is included in the *OSF/1 Security Features Administrator's Guide*.

In contrast to command authorizations, the system defines a set of kernel authorizations, or override authorizations, associated with specific kernel actions that are allowed to privileged users. These authorizations control the ability of trusted commands to override basic system constraints, affect

the way that a user can enable privileges for all commands he or she executes, and limit the privileges that a user can associate with a program.

Kernel authorizations are distinct from command authorizations in that kernel authorizations are used to enable specific security policy overrides in certain trusted applications, while command authorizations signal trusted commands to grant the user the requisite operational rights. Kernel authorizations are directly related to the kernel-recognized privileges; command authorizations are defined as needed to identify categories of operations that an administrator may want to grant to some subset of the user community on a system.

From an implementation standpoint, kernel authorizations are associated one-to-one with the defined privileges. In fact, they are stored in the same data structure used to store privilege sets. The kernel participates in storing and manipulating privilege sets. The command authorizations are defined and maintained in databases outside the kernel. Differences between types of command authorizations are purely conceptual; the authorizations are stored in a bit vector data structure and tested individually.

The ISSO designates a program as trusted at installation time by assigning privileges to it or designating it to run with superuser compatibility. When the ISSO assigns privileges, the program can make use of only those privileges assigned to it combined with those inherited from the process in which it runs. When the ISSO designates superuser compatibility, all but a few privileges are available to the program.

A user is considered trusted to the extent that the ISSO has assigned one of the following to his or her authentication profile:

- Command authorizations that are honored by trusted applications
- Kernel authorizations that allow the user to transmit privileges to every program he or she runs

The following privilege sets are defined for each process:

Base privilege set (BPS)
: Contains the set of privileges that are automatically enabled when the process executes a new program. This set normally holds mode privileges that control functions that are unrestricted on a traditional UNIX system, for example, the ability to create and execute SUID programs.

Kernel authorization set (KAS)
: Contains the set of privileges the user responsible for the process is authorized for. They control the ability of a process to add privileges to its base privilege set. They also control the ability of a process to assign privileges to a file. In addition, several administrative commands consult this set to determine whether or not to enable privileges that cause the kernel's access control policies to be bypassed.

Effective privilege set (EPS)
: Contains the privileges the kernel makes privilege comparisons against. A trusted program implements the principle of privilege bracketing by adding and removing privileges from the effective privilege set before and after the operations that require them. (Trusted programs also enable *only* required privileges, in keeping with the least privilege principle that subjects should be given no more privilege than is necessary to enable them to do their job.)

Note that the elements of each of these sets correspond to the privileges that the kernel enforces.

In a system configured with **SEC_PRIV**, trust is assigned to application programs by virtue of the contents of the two privilege sets associated with executable files:

Potential privilege set (PPS)
: The set of privileges a program is trusted to enable

Granted privilege set (GPS)
: The set of privileges automatically placed into a process's effective set when it executes the file as a program

The potential set determines which privileges a process can acquire by executing the file. A file starts out with an empty PPS, and the PPS is cleared upon every write to the file. A process with the **chpriv** privilege and owner rights to a file can change the file's PPS to include any of the privileges the process is authorized for (that is, any in its KAS).

When a user executes a program provided by another user, each user brings a set of privileges to the action. The owner of the process and the owner of the executable file each provide an initial set of privileges that form the effective set of the joint venture. This effective set can be enlarged, subject to constraints provided by both parties. Privileges can be added to the effective set that are in either the potential set or the base set. The base set

can itself be enlarged, but only by the addition of privileges that are in both the kernel authorization set and the potential set. This protects both parties in the event that another file is executed. The base set used with this new file would contain only those privileges allowed by both parties. This technique prevents privileges from being combined in unforeseen ways.

A file's granted privilege set determines which of the privileges from the PPS should be automatically activated (put into the EPS) when a process begins executing the file. The GPS is generally used with old programs (often programs for which the source is no longer available) that require privileges to perform their functions, but have not been modified to enable and disable privileges as necessary. The GPS of a newly created file is empty, and is cleared on each write to the file. In order to change the GPS, a process must own the file (or have the **owner** privilege) and have the **chpriv** privilege as well. Only privileges in the file's PPS can be added to its GPS. Removing a privilege from the PPS implicitly removes it from the GPS.

The following example demonstrates a design decision of OSF/1 authorizations and privileges. Often it is desirable that a user be able to do some things that require a certain privilege, but not others. For example, the operator should be able to read everyone's mail file as part of the backup process, but not be able to read it for general snooping purposes. Rather than give the user the **allowdacaccess** kernel authorization or base privilege, which would allow unlimited use of the privilege, it is better to give the backup program the **allowdacaccess** potential privilege, and have the backup program enable that privilege only if the user has the **operator** command authorization. In effect, command authorizations are a decoupling of what is done at a high level and what privileges are required to do that task.

For compatibility, programs can also be designated as trusted by making them SUID to **root** and placing the **sucompat** privilege in the program's potential and granted privilege sets. (The **supropagate** privilege will cause **sucompat** to be added to the potential and granted privilege sets of any file that the program makes SUID to **root**.) A system without file-based privileges treats all programs that are SUID to **root** as trusted in this fashion.

The kernel's decision on whether to grant a process a privilege is based on the following criteria:

- The effective privilege set of the process

- The privilege that was requested
- If the program is running with **sucompat** in its effective set.

The privilege decision is based on whether the process is operating in superuser compatibility mode (**sucompat** is in the process's effective privilege set, or **SEC_PRIV** is not defined). If it is not, the privilege computation merely checks the process's EPS. Otherwise, the privilege is categorized into one of the following categories:

- Privileges that are granted according to the effective set only
- Privileges that are always granted if the process EUID is 0 (zero), and are not granted otherwise
- Privileges that are granted if the process has EUID 0 (zero) or if the privilege appears in the process's effective privilege set

The discretionary logic is encapsulated in the **privileged( )** kernel routine. The **privileged( )** routine also stores the fact that a privilege was used in the per-process privilege use vector that is stored in the audit record for the system call at audit production time.

Library routines are modified or designed to support the least privilege principle in trusted applications. Each routine runs with as few rights as necessary to accomplish its task. These library routines are used to implement a protection philosophy based on a program's enforcement of who is allowed to use it and the privileges it needs to accomplish its task.

15.7 Security Administration

Security administration is split into three roles:

- The ISSO sets system defaults for users, maintains security-related authentication profile parameters, modifies non-ISSO user accounts, administers the audit subsystem, assigns devices, and ensures system integrity.
- The system administrator creates user accounts, modifies ISSO accounts, creates and maintains file systems, and recovers from system failures.
- The operator administers line printers, mounts and unmounts file systems, and starts up and shuts down the system.

15.8 The Discretionary Access Control Policy

Discretionary access control (DAC) allows users to specify who has access to their files. Traditional UNIX has a simple form of DAC that allows users to specify access to a file based on whether the user requesting access is the file owner, a member of a group associated with the file, or any other person. Under this system, file owners selectively grant read, write, and execute permission to users in the three categories. The drawback is that a user can only broadly specify who has access to his or her files. Users cannot easily allow or disallow file access to individual users.

If configured with an access control list (ACL) mechanism, the system improves upon traditional UNIX DAC. In addition to traditional owner/group/others access control, each object may also have an ACL. The ACL specifies the permissions given to individual users or groups of users. Section 15.8.2 describes ACLs in detail.

15.8.1 Discretionary Access Contol Components

The DAC policy has the following components:

- The DAC policy module
- The DAC policy daemon
- System calls
- Library routines
- Commands

The DAC policy module implements the entry points for the DAC entry in the security policy switch. The DAC policy allocates one tag for subjects and two tags for objects (the second object tag holds the default ACL for directories). The system maps between the subject tag and a user ID, group ID, and supplementary group list. The DAC policy daemon stores (in the security policy database) the mapping between the subject tag stored in the tag pool and the subject internal representation.

The object tag maps to a data structure containing the owning and creating user and group of the object, the object's permission bits, and the object's ACL. For System V IPC objects, the owning user, creating user, owning group, and creating group are the same as those stored in the IPC data

structures. For other objects, the owning user is equal to the creating user and the owning group is equal to the creating group. The users, groups, and modes must be contained in the object internal representation because

- The DAC module caches discretionary access decisions between subjects and objects that depend on the UNIX discretionary attributes and the ACL.
- ACL entries may contain owner and owning group specifications. The daemon process must know the owner and group of the object to properly match against ACLs containing such specifications.

The ACL is a variable-length list of ACL entries. The DAC module supplies the owning and creating user, owning and creating group, and permission bits when requesting that the daemon map an internal representation to a tag.

The DAC policy module interacts with the DAC policy daemon through the security policy device to

- Make ACL decisions between subjects and objects
- Retrieve the ACL for an object
- Set the ACL on an object
- Interact with the daemon when the owner, group, mode, or ACL on the object changes
- Map subject identities to subject tags

The DAC policy daemon maintains the database of mappings between object and subject tags and internal representations. The daemon also makes the discretionary access decisions between a subject and the UNIX DAC attributes and ACL on an object. The daemon interacts with the security policy module through messages exchanged through the security policy device.

Each system call that has a permission bit check in the unsecured OSF/1 operating system includes an ACL check in systems incorporating the ACL policy. The system defines new system calls to manipulate object ACLs. These system calls are actually library routines layered on the **getlabel( )** and **setlabel( )** policy-independent system calls.

The system provides library routines which map between internal and external representations of ACLs. A set of routines which communicate with the daemon to map between tags and internal representations is also available.

At the command interface, the system provides a set of utilities that set and retrieve ACLs on objects. In addition, there is a screen-oriented interface tool that allows users to associate ACLs with files and to test access to an ACL given a particular user and group ID.

15.8.2 Access Control Lists

An ACL for a file identifies the individual user or group(s) of users who may access the file. The file owner maintains the file's ACL using additional commands provided with the system. The POSIX ACL mechanism defines an upward-compatible additional protection mechanism for system objects. The mechanism enables objects protected with traditional UNIX DAC to work as before, while also enabling traditional UNIX DAC to be viewed as a simple three-entry ACL. In addition, a mask entry definition allows the POSIX ACL mechanism to satisfy the POSIX requirement that an additional protection mechanism only makes access more restrictive than that implied by the UNIX DAC attributes on an object.

An ACL is an ordered list containing three or more ACL entries. The three base entries, the *owner* entry, the *owning group* entry, and the *other* entry, correspond to the UNIX DAC attributes. An extended ACL entry additionally contains a *mask* entry (required if the ACL contains more than three entries), and zero or more *additional user* and *additional group* entries. Each entry specifies a matching condition and a set of permissions. The entry consists of an ordered pair containing

Identity part

Specifies process identity criteria:

- Type (owner, owning group, or other)
- Qualifier (user ID, group ID, or NULL)

Permission part

Specifies the access permissions the process matching the criteria has to the object

To determine whether a particular user can access a particular object, the system scans the object's ACL. The ACL is searched in a specific order, and matching entries are used to construct the permissions that are allowed to the process. A process matches the owner entry if the effective user ID of the process matches the owner of the object (or creator if the object is a System V IPC object). A process matches the group entry if the effective group ID of the process or any of the process's supplementary groups matches the owning group of the object (or creating group if the object is a System V IPC object). A process matches an additional user entry if the effective user ID of the process matches the qualifier in the entry. A process matches an additional group entry if the effective group ID of the process or any of the process's supplementary groups matches the qualifier in the entry.

The access decision against an ACL is as follows:

1. If the user ID of the process matches the user ID of the owner entry, access is granted if the permissions are sufficient in the matching entry; otherwise, access is denied.
2. If the process matches any of the additional user entries, access is granted if it is allowed by the permissions in the matching entry *and* in the maximum permissions specified by the mask entry (if one exists), and is denied if it is not.
3. The system forms a permission set that is the union of all matching group and additional group entries. If the permissions requested are in this union *and* in the mask entry (if one exists), access is allowed; access is denied if they are not.
4. Otherwise, access is granted if it is allowed by the permissions in the *other* entry, and denied if it is not.

If an object does not have extended entries in its ACL, it has a WILDCARD ACL. The notion of a WILDCARD ACL is not visible at the user or programming interface, but is specified in the design to indicate that the object has a WILDCARD tag for its access ACL in the tag pool associated with the kernel data structure that describes the object. When there is a WILDCARD ACL, the kernel does not need to consult the daemon for access checks. Access decisions for an object with a WILDCARD ACL are based totally on traditional UNIX DAC criteria, behavior that is backward-compatible with traditional UNIX behavior. If the object has a non-WILDCARD ACL, access decisions for the object are based on the entries that appear in the ACL (base and extended entries).

15.8.3 Discretionary Access Control Privileges

The extended OSF/1 system divides override privilege with respect to UNIX DAC into the following defined privileges:

owner Overrides ownership checks. A process with the **owner** privilege may act as the owner of any object.

allowdacaccess
Overrides checks against the object's permission bits and ACL, and guarantees that a discretionary access check (both against the UNIX discretionary attributes and the ACL) always succeeds.

lock Allows a process to set the sticky bit on nondirectory files. In addition to the **lock** privilege, a process must have ownership rights to a regular file to set its sticky bit. To set the sticky bit on a directory, the process must merely have ownership rights.

chown Allows a process to change the file owner or to change the file group to one not equal to the process's effective group ID or one of the process's supplementary groups.

chmodsugid Allows a process to set the SUID and/or SGID bits on files (not required if indicating mandatory locking).

setprocident Allows a process to set the SUID bit on files whose owning user is different from the process's effective user ID, or to set the SGID bit on files whose owning group is different from the process's effective group ID or any of the process's supplementary groups.

15.8.4 ACL Representations

The DAC policy allocates two tag pool slots. The first tag pool slot is used for the object's access ACL, and the second slot is used for the object's default ACL. The default ACL tag pool slot is only used if the object is a directory. If an object is protected only by traditional UNIX DAC (corresponding to the three base ACL entries), the object has an access ACL tag value of **SEC_WILDCARD_TAG_VALUE**. Otherwise, the tag value for the access ACL maps to a creating user and group and a list of ACL entries through the security policy database maintained by the DAC security

policy daemon. If a directory has a default ACL, its tag maps to the default ACL binary representation through the security policy database. If not, or if the object is not a directory, the default ACL tag slot contains a **SEC_WILDCARD_TAG_VALUE** tag.

ACLs have internal and external representations outside of the kernel. The internal representation of an ACL is used by user programs. It contains the identity and permission parts of an ACL entry. The external representation of an ACL is the representation specified by the user when defining the ACL, and is the representation printed by commands when listing the ACL associated with objects. Library routines convert between internal and external representations.

15.8.5 Example: Changing an ACL

Assume that a process wants to change the ACL associated with a file. The process calls the **chacl( )** system call (which is actually a library routine, a veneer to **setlabel( )**, the actual system call entry point), specifying the filename, a pointer to the array containing the ACL structures, and the size of the array. The **chacl( )** call specifies the security policy switch index for the discretionary policy and an offset of 0 (zero) in the tag pool relative to that policy's tags. The **chacl( )** call calls the **setlabel( )** system call, specifying the switch index, the filename, and the internal representation of the new ACL (specified as a buffer and a length). The **setlabel( )** call calls through the security policy switch entry for the discretionary policy to check that object-specific constraints have been satisfied (the **sw_setattr_check( )** switch function).

The security policy module copies the attribute's internal representation from user space into a message and sends it to the security policy daemon through the security policy message driver. The daemon looks up the internal representation in its database, allocating a new tag and creating a new mapping if no existing entry is found.

The daemon puts the tag into a response message, and sends it back to the waiting module through the security policy driver. The module returns the new tag and a success code back to the **setlabel( )** system call, which then calls the discretionary module through the **sw_setattr( )** switch function to apply the new tag to the object. The module sets the appropriate tag pool slot to the new tag and returns success to the **setlabel( )** routine. The **setlabel( )** call, in turn, returns success to the user process.

Note that the policy module and the message driver merely act as conduits for the internal representation. The daemon actually validates the format of the internal representation and allocates a tag for it. Note also that all attribute changes must pass through the policy daemon. Only the daemon knows how to convert an internal representation into a tag.

15.9 Mandatory Access Control

Under MAC, all information is classified according to how sensitive that information is. This classification is called a *sensitivity level*. Users are rated according to the maximum sensitivity of information they are cleared to handle. This rating is known as a *clearance*. The system decides who can access what information based on the information's sensitivity level and the user's clearance. Only users with sufficiently high clearances can access information of certain sensitivity levels. A user who wants to open a file for reading must have a clearance that is at least as high as the file's sensitivity level; that is, the subject must dominate the object (no "read up").

Similarly, the system must prevent a process from reading information and writing it to an object that can be seen by users at a lower sensitivity level than the information (no "write down"). This comparison of sensitivity level and clearance is called a multilevel security model. (Access must also be granted by the system's discretionary access controls.) User access to objects is mediated by the system and cannot (normally) be overridden by ordinary users.

Each subject and each object is assigned a sensitivity label when it is created by the system. Subjects and objects generally inherit their sensitivity labels from the processes that create them. The system administrator can permit users to choose their sensitivity levels when they log in (as long as the level chosen is equal to or less than that user's clearance). The user might choose a level below his or her clearance to ensure that programs inheriting this sensitivity label have limited access to privileged information.

The system compares subject and object sensitivity labels in making an access decision. In keeping with the Orange Book, a sensitivity label is composed of a hierarchical classification and a nonhierarchical set of categories (or compartments).

An object's sensitivity label denotes the sensitivity of the object's contents and attributes. A subject actually has two labels, a sensitivity label and a clearance label. One denotes the sensitivity level the process is currently executing at, and the other denotes the clearance level of the user on whose behalf the process is running. (Note that users can only choose to execute processes at a sensitivity level dominated by their clearances.)

Sensitivity labels and clearances have an external representation, which includes a classification name followed by a comma-separated list of categories enclosed in slashes. This form of label is called the expanded form of the label. On input, a user can specify synonyms to simplify entry of long names or long lists of categories. The ISSO defines synonyms at a given site using the **mandsyn** utility. The internal representation is used by the programming interface and by the mandatory policy daemon.

15.9.1 Mandatory Access Control Components

The MAC policy has the following components:

- The MAC policy module
- The MAC policy daemon
- System calls
- Library routines
- Commands

The MAC policy module is a collection of routines called through the MAC policy entry in the security policy switch. A policy module for each policy implemented on the system is linked into the kernel. Functions in the MAC policy module exist to perform operations such as

- Computing access decisions between subject and object sensitivity label tags. The module may have to consult the MAC policy daemon to compare the internal representations associated with the tags and return an access decision.
- Setting and retrieving the process sensitivity label or clearance.
- Setting and retrieving the object sensitivity label.
- Enforcing privileges.

15.9.2 MAC Privileges

Privileges allow a process to bypass some or most of the limitations imposed on normal users by the MAC policy. The MAC policy module access routine checks whether the process requesting an operation on an object has a privilege (in its effective privilege set) relevant to the operation. If so, the MAC policy module grants the access request without checking the sensitivity labels of the process and the object. For example, a process with the **allowmacaccess** privilege returns success to most MAC checks.

Processes need **writeupsyshi**, **writeupclearance**, or **allowmacaccess** privileges to upgrade an object, and **downgrade** or **allowmacaccess** privileges to downgrade an object. If a file is downgraded, its new level must continue to dominate the parent directory's level. Otherwise, the downgrade is not allowed.

In an unsecured OSF/1 system, certain directories (such as **/tmp**) are writable by all users. On a secure system, the ability of processes at different sensitivity levels to write into a common directory would be a violation of MAC. Therefore, such a directory is implemented as a multilevel directory, which contains child directories for each sensitivity level. A process needs the **multileveldir** privilege to see the child directories within the multilevel directory. When a process without the **multileveldir** privilege references the multilevel directory, it is automatically diverted to the child directory with the same sensitivity level as the process. If no child directory exists at that level, one is created by the system.

For example, various programs such as the C compiler use the **/tmp** directory to hold their temporary files. Even though the contents of these files may be securely protected, the existence of a temporary file as well as its name could be easily discovered by anyone who performed an **ls** on the **/tmp** directory. Such information about Top Secret temporaries, for example, should not be known to those operating in the Secret domain.

One approach to dealing with this problem might be to change all applications so that they do not use public directories, but instead find directories at appropriate security levels. A better approach is to deal with the problem transparently. A directory such as **/tmp** may be set up as a multilevel directory. It is then transparently split into a number of subdirectories, one for each security classification. Thus, a reference to the file **/tmp/foo** by a process executing as Top Secret would be translated into a

reference to the file **/tmp/mac***xxxxxxxxxxx***/foo**. The name of the multilevel child directory is always **mac** followed by an ASCII representation of the decimal value of the tag that corresponds to the sensitivity level of the process. Leading zeroes are prepended to the decimal value to pad it to 11 characters (long enough to store any 32-bit number in decimal, yet short enough to not exceed the 14-character filename limit of some file system types).

The diversion to the child directory is as transparent as possible, while still maintaining the desired separation of files at different sensitivity levels. For example, if **/tmp** is a multilevel directory, a **cd /tmp** by a process with a System Low sensitivity level and without the **multileveldir** privilege enabled will divert the process to **/tmp/mac00000000002**, but a **pwd** will show the user's current directory to be **/tmp**. A **cd ..** will return the user to / (automatically diverting through the multilevel parent directory that was skipped by the original **cd** command).

The diversion works the same way for processes at other sensitivity levels, except that they are diverted to a different multilevel child directory (**mac00000000003** for System High processes, for example).

Explicit creation and regrading of subdirectories are not allowed because there is a unique mapping from subdirectory name to sensitivity level. Subdirectory diversion and automatic multilevel child directory creation is implemented by hooks implanted in the filename lookup routines. Conversion to multilevel directories is handled through new system calls that call routines specific to each file system type that set and clear the multilevel directory security attribute.

A common use of the **multileveldir** privilege is to allow all of the subdirectories of a multilevel directory to be accessed easily for purposes of making backups of a file system. Because processes are automatically diverted to a subdirectory if they do not have this privilege enabled, it would be difficult to try to do full backups without this privilege enabled.

If the process has **sucompat** in its effective set, and has effective UID 0 (zero), it is granted all of these privileges, except **multileveldir**. That privilege is granted only if it appears in the effective privilege set.

15.9.3 MAC Access Decisions

When making MAC checks on behalf of system calls, the system grants the process MAC read access to the object if the sensitivity level of the process dominates the sensitivity level of the file or if the process has the **allowmacaccess** effective privilege.

The process has MAC write access to the object if any of the following is true:

- The sensitivity level of the process is equal to the sensitivity level of the object.
- The sensitivity level of the object dominates the sensitivity level of the process and is dominated by the process clearance, and the process has the **writeupclearance** effective privilege.
- The sensitivity level of the object dominates the sensitivity level of the process, and the process has the **writeupsyshi** effective privilege.
- The process has the **allowmacaccess** effective privilege.

The system administrator defines two system-wide and frequently used sensitivity labels: System High and System Low. These are defined in the configuration file **/etc/policy/mac/config**. System High dominates System Low, and the two form a range that absolutely bounds activity on the system. The TCB code that enforces the MAC policy contains a number of optimizations that take advantage of this fact. For example, when making a MAC access decision, the kernel tests the tags in question against the well-known tag values **SEC_MAC_SYSLO_TAG** and **SEC_MAC_SYSHI_TAG**. These extra tests often allow the kernel to avoid the overhead of consulting the MAC policy daemon for an access decision.

15.9.4 MAC System Calls, Library Routines, and Commands

The **getlabel( )** and **setlabel( )** system calls that address the MAC attributes on subjects and objects call through the security policy switch to perform MAC-specific label setting and retrieval functions. Except for the multilevel directory calls, all MAC function calls retrieve or set a sensitivity

label or clearance on one of the system's subject or object types. These function calls pass either **getlabel( )** or **setlabel( )** to the appropriate tag offset.

Library routines for MAC convert between internal and external representations of sensitivity labels, communicate with the policy daemon to map tags and make decisions, and access the subdirectories of a multilevel directory. In performing these activities, they must at times make calls to the kernel (through system calls) to operate on subjects or objects, or make calls to the daemon to retrieve sensitivity label internal representations.

In addition to existing UNIX command changes to implement MAC, there are a few new, specific MAC commands. Existing commands that are affected are the utilities that

- Adjust the sensitivity level at login time
- Access the raw UNIX file system structure
- Deal with multilevel directories
- Deal with users who modify a system database but may be logged in at a different sensitivity level than the database (for example, **passwd**)
- Print information about subjects and objects in the system that may be at different sensitivity levels.

New MAC commands have been created to

- Print the current process's sensitivity label and/or clearance to standard output (**getlevel**).
- Change a file's sensitivity level. The resulting file sensitivity level depends on the user's authorizations and the file's sensitivity level. If the operation is a downgrade, the file must continue to dominate the sensitivity level of the file's parent directory. This command produces an audit record for all outcomes (**chlevel**).
- List the sensitivity level of files and directories (**lslevel**).
- List and replace sensitivity labels on message queues, semaphores, and shared memory segments, depending on the arguments supplied (**ipclevel**).

15.9.5 MAC Database Protection Principles

This subsection describes database protection in terms of MAC attributes. Section 15.10 expands the protection discussion to include all attributes.

All protected databases are labeled with the System Low sensitivity level. The following principles are used to protect the system files critical to the operation and security of the system.

Protected password database
: Contains user clearances in authentication profiles. Each profile is kept in a separate file, and each is protected at System Low. When a user changes his or her password, the **passwd()** program creates a new file to store the modified database. This new file inherits the sensitivity label of the creating process. If left at that level, the database might be improperly protected or inaccessible to user processes that should have access to it. To remedy this situation, many commands enable the **allowmacaccess** privilege. After updating the appropriate database, the programs change the sensitivity label of the database back to the appropriate level. This protection mechanism is used for all the databases described in this section.

System defaults database
: Stores default sensitivity level ranges, the default user clearance, and the single-user-mode sensitivity level. This database is protected at System Low.

Device assignment database
: Stores the default device sensitivity level range and the default single-level sensitivity level. This database is protected at System Low.

File control database
: Describes the attributes of security-relevant files, stored at System Low. The only sensitivity levels stored in this file are the strings **syshi**, **syslo**, and **WILDCARD**. These do not convey the classified compartment names.

Audit control files
: Binary files that are read by the audit subsystem. These are stored at System Low because they do not contain literal compartment names.

Audit output files
: Contain audit records corresponding to all processes running on the system. These files, and the directories in which they are stored, have System High sensitivity labels.

Audit daemon
: Must be run at System High.

System accounting file
: Has information about all processes in the system. It has a System High sensitivity label.

15.10 Authentication Subsystem and the Security Databases

The process of verifying the identity of the user is called authentication. The system authenticates each user at login time and when a user attempts to change identity through the **su()** program. The authentication subsystem uses and maintains the security databases related to authentication that contain user parameters and statistics for the system, terminal, and user.

OSF/1 offers the following enhancements to traditional UNIX password mechanisms:

- In traditional UNIX systems, any user can choose his or her own password. In secured OSF/1, the ISSO must determine site and per-user password selection methods. Default settings are provided with the system, but the ISSO at a site can relax or tighten these restrictions.
- Many UNIX implementations do not impose restrictions on the chosen password, checking only for length, resemblance to the login name, and that they are purely alphabetical. Secured OSF/1 additionally supplies checks for palindromes, resemblance to user names on the system, and English words.
- In traditional UNIX systems, encrypted passwords (stored in a central file) are visible to all users. The encryption algorithm (**crypt()** is also commonly available. Thus, a penetrator can apply heuristics to guess passwords. Secured OSF/1 stores encrypted passwords in protected (not public-readable) files. Secured OSF/1 also provides assurance that the cleartext of the password will not be compromised, by clearing buffers that store the cleartext immediately after use.

- Traditional UNIX systems allow accounts without passwords. Secured OSF/1 allows the ISSO to require passwords of all accounts.
- Password aging is not a requirement. Accounts with this feature disabled are never forced to change passwords. Secured OSF/1 enforces a password lifetime, after which the account is disabled.
- The UNIX implementation of aging does not satisfy *Green Book* requirements. Secured OSF/1 requires that when a password is created or changed, a time period exists when the password cannot be changed again. This prevents a user from changing a password multiple times, ultimately back to the original password, as a way to avoid a true password change.
- Passwords are significant only to eight characters. Thus, pass phrases are impractical in traditional UNIX systems. A pass phrase is like a password, but it consists of several words instead of just one. Thus, it is harder to guess. Secured OSF/1 includes a routine (**passlen**( )) that computes a minimum password length based on parameters stored in the security databases. It also allows the ISSO to choose a maximum password length. The maximum length of a generated password is 40 characters, but this can be extended or reduced by changing the definition (**AUTH_MAX_PASSWD_LENGTH**) and recompiling.

For each process, the system maintains a login user ID (LUID). The LUID identifies the user associated with a process. Once the LUID is initialized (by the authentication program, or the **su**( ) or **epa**( ) program, or by the daemon itself for system daemons), it is never changed, regardless of authorization or identity. It is thus immutable. The LUID is necessary because users can acquire **setprocident** effective privilege and can change their process's real user ID and effective user ID. In such cases, auditing is based on the LUID, not on the effective user ID. Authorization checking is always done against the process's LUID.

15.10.1 Authentication Database

The heart of the authentication subsystem is the set of security databases that store security parameters enforced by the system. These are the following:

- Protected password database
- System defaults database
- Terminal control database
- File control database
- Command authorizations database
- Device assignment database

All of the databases have similar formats, although each has a different purpose. The low-level routines to read the entries and the fields in each entry are shared among the databases.

15.10.1.1 Protected Password Database

The protected password database stores the encrypted password for the user's account, as well as all account-specific parameters (referred to as the user's authentication profile). Any parameters not specifically set in the user account protected password entry are set by default to the corresponding value in the system default database, described in Section 15.10.1.5.

Each user's protected password database entry is stored in a separate file. This makes looking up each user's entry faster, because scanning a large file is avoided, and updating a user's entry does not require that all accounts be locked for the duration of the update. A directory hierarchy exists under **/tcb/files/auth** that is structured like terminal descriptors in the **/usr/shared/lib/terminfo** database. Each user's entry is contained in a separate file whose name is the user's account name. The file resides in a directory in **/tcb/files/auth** whose name is the first letter of the account name.

Both the **/tcb/files/auth** directory and its subdirectories are protected with owner and group **auth**. Directories have mode 0770, while files have mode 0660. Programs that access the database are SGID to **auth** or enable the **allowdacaccess** effective privilege. The protected password database files

have a sensitivity label of System Low. Programs that access or update the database need to enable appropriate privilege to be able to read or update the database depending on the sensitivity label of the program accessing it.

Each entry in the protected password database corresponds to a single user. The login name is the primary key. The secondary key, the UID, is a cross-check to the corresponding entry in **/etc/passwd**. All programs dealing with allocation of users ensure that

- Each account is assigned to one real user.
- Each account has one login name.
- Each account has one reference UID.
- For each login name, there is a single UID and a single database entry.

A lock on an account prevents anyone from logging into the system under that account. With the account locked, the next time the user gives his or her user ID and password, the system responds with a message stating that the account is locked. Only the ISSO can unlock the account.

There are three types of locks:

- Unconditional (administrative lock)
- Maximum number of login retries exceeded
- Password lifetime has expired

A field in the protected password database entry for the account stores the status of the administrative lock. Another field contains the maximum number of login retries allowed, and another the password lifetime.

The system authentication program and **su**( ) enforce the lock conditions. In addition, they do not allow access to accounts that have been retired. The lock condition is reversible; once retired, an account cannot be reinstated (unless the protected password database entry is edited manually). The **su**( ) program checks if the destination account is locked and prevents the transition to that account if it is. All locks are audited.

All daemons that create processes that run on behalf of users enforce this behavior. A library routine is supplied that enforces the lock conditions.

The successful and unsuccessful login counts are kept with the protected password data. Thus, the system remembers breakin attempts across sessions and system reboot.

A user can lock another user's account by simply making repeated failed attempts on that account. However, audit information can help track down the terminal where the attempts are made. (One safeguard against locking the whole system is that the **root** account on the system console cannot stay locked. It can be locked by repeated failed attempts; however, the next login attempt from the console will print out a warning of the lock, log an audit record, clear the lock, and let the login succeed.)

15.10.1.2 Terminal Control Database

The terminal control database stores a threshold of the number of unsuccessful login attempts allowed at a terminal before that terminal is considered locked.

There is a delay field that controls how quickly the system allows a new login attempt after a failed attempt. The database also stores the user ID and time of the last successful and unsuccessful login attempts at that terminal, and of the last session terminated. These fields can be used to detect penetration attempts.

The system associates an administrative lock and a count of unsuccessful attempts with each terminal.

The terminal control database stores a terminal-specific delay factor that spreads out login attempts. (The system-wide delay factor is stored in the system defaults database.) This feature impedes automated attempts to guess authentication information. The default delay for the system is used as input to determine the minimum size of a password. The delay may be set to 0 (zero).

Each terminal entry has the times and user ID (if the name has an associated **/etc/passwd** entry) of the last attempts (successful and unsuccessful) to log in, and the last logout. The terminal control database stores the user and time of the last successful and unsuccessful login attempts and last logout from a given terminal.

The terminal control database has shorter entries than the protected password database and is updated infrequently. It is stored in a single file, **/etc/auth/system/ttys**. The terminal entry is updated on each successful and unsuccessful login attempt, and on each session termination. The authentication program does not allow a user to log into a terminal unless there is a valid (and unlocked) terminal control database entry. The

exception is the **root** account, which is always allowed access on the system console, even if the databases are corrupt.

15.10.1.3 File Control Database

The file control database contains information about security-relevant files, each of which has an entry in this database. With the information in these entries, the system administrator can possibly detect tampering on security-relevant files. This helps thwart a penetrator who has managed to overwrite a system file with a Trojan Horse file.

The file control database is accessed by the **integrity** program, which checks each entry in the database against the corresponding file in the file system. The **setfiles** program not only checks the entries, but fixes them. A **setfiles** with no command line arguments fixes all files in the file control database. A list of files can also be specified. The file control database is also accessed by the **create_file_securely**( ) library routine, which sets the security attributes (mode, owner, sensitivity level, privileges, ACL, and group) of each newly created security-relevant file. Like the terminal control database, the file control database is contained in one file, **/etc/auth/system/files**.

The file control database may have a filename whose last component is * (asterisk). This entry matches all files in the corresponding directory and may be used as a wildcard entry when all files in a directory have the same attributes. The entries in the database that precede the wildcard entry and refer to files in the directory named by the wildcard entry override the wildcard entry. Programs that process the database remember the files they have accessed and process each file only once.

Only static file attributes should be present in the file control database entry. For example, some programs may require that some system files have a specific group identification, but the user owning the file may vary (for example, mailboxes in the public mail directory). Absent file control database entry fields (for example, user ID) successfully compare with all values of the corresponding file attribute.

15.10.1.4 Command Authorizations Database

A user's command authorizations are stored in the user's authentication profile, which is not generally readable. Programs that must check user command authorizations query the publicly readable command authorizations database to determine the authorizations a user possesses. The programs that update a user's authentication profile update the command authorizations database whenever they change a user's command authorizations.

The system is designed to accommodate additions to the command authorizations the system recognizes. The list of authorizations supported at a site is determined by the contents of the command authorization definition file. This file contains a list of authorizations and the relationships between them. Command authorizations are hierarchical; possession of certain authorizations implies possession of others.

The **authorized_user**( ) routine checks whether a user has a specified command authorization. It understands the hierarchy of authorizations as defined in the command authorizations definition file (**/etc/auth/system/authorize**). All command authorizations are checked against the login user ID. Thus, a user does not gain a new set of command authorizations across **su**( ) transitions or by running SUID/SGID programs.

The **hascmdauth**( ) routine checks the presence of an authorization in a specific authorization vector. This routine is useful for checking a specific user's authorizations against the user's protected password entry or for checking authorizations within a program that has not yet set its LUID. **authorized_user**( ) returns TRUE for all authorizations when called from a program without an LUID.

15.10.1.5 System Defaults Database

The protected password, device assignment, and terminal control databases allow for system defaults. The system defaults database stores default values for database fields. The default fields are consulted if the corresponding field in the user-specific, device-specific, or terminal-specific database is not set.

In its external form, the database stores fields with the same names as the terminal control, device assignment, and protected password databases.

Database access routines return the system default values for each database with each record returned. Thus, there are field and flag structures for the user or terminal-specific entry and the system default values. In the internal form, each database structure holds specific and default field structures, and specific and default flag structures. The default database structure has field and flag structures for the protected password, terminal control, and device assignment databases. The system defaults database consists of only one entry, keyed by the name **default**. The database is stored in the file **/etc/auth/system/default**.

The system defaults database is fairly static. It is updated only by the ISSO. This database is not written as a side effect of user operations.

15.10.1.6 Device Assignment Database

The device assignment database stores device name synonyms and import/export restrictions. In OSF/1, it is possible for several special device pathnames to reference the same physical device. When such a device is reassigned to another user session, all references to the physical device must be invalidated. The device assignment database entry stores a list of pathnames for the physical device. It also stores a device type and the external name of the device. The database is stored in the file **/etc/auth/system/devassign**.

15.11 Audit Subsystem

The audit subsystem collects information on all security-related system activity. The subsystem writes this data to an audit trail. The time from the enabling of audit (automatic or administrator-initiated) to the corresponding disable (which may occur as a by-product of system shutdown) is called an audit session. Each audit session is identified by a unique session ID. The audit subsystem increments the session ID each time it enables auditing. The data that the audit subsystem collects during a single audit session is called the session audit trail. The session audit trail for the current session is often simply referred to as the audit trail.

Each session audit trail is composed of a sequence of audit records. An audit record stores the information required to identify one security event.

Together with the records that precede the record in the audit trail, a program can identify the user accountable for an action, the object or objects affected, and the specific action performed.

Audit records are categorized by event type. The ISSO can select the defined event types when specifying which audit records the audit subsystem should pass to the audit trail and which records the audit subsystem should print during analysis. The selection criteria can include upper and lower bounds on the sensitivity level of a process, as well as lists of user IDs and group IDs to audit. By default, all sensitivity levels and user/group IDs are audited.

The TCB generates audit records. Both the kernel and trusted processes generate audit records; the kernel records the actions of system calls made by processes to request changes to subjects and objects that the kernel maintains, while trusted processes record the actions of the users that invoke them. A kernel audit record stores the parameters and success or failure of a specific system call (such as opening a file for reading and writing), while a trusted process audit record stores the higher-level actions of a trusted program (such as the use of a command authorization).

The following design decisions were made for the OSF/1 audit implementation:

- The system stores a binary audit record format. Binary structures are simpler and faster to generate and manipulate and consume less disk space. When an audit transfer capability is required (to print reports on a different machine than the system on which the data is generated), or a standards organization decrees a new format, simple tools can be developed to convert the current audit record format to the desired format.

- The audit subsystem records relative pathnames in audit records, recreating full pathnames at reporting time. This saves the time overhead associated with preserving pathnames in the kernel for current and **root** directories and open files, and the space overhead of the additional data structures in the kernel and the extra space for absolute pathnames in the audit trail.

- Not all of a process's attributes are stored with each audit record. Specifically, the process identity (effective and real user and group IDs, login user ID, and supplementary groups) is not recorded in each audit record, but rather is stored when the process is first created and when each of these parameters changes.

- Some events must be audited to maintain the process state. The cost of generating the additional audit records to maintain process state is far less than that of increasing the size of audit records to store full pathnames, and the space and time overhead of maintaining process state within the running operating system.

15.11.1 Audit Subsystem Components

The audit subsystem is composed of a set of modules that are added to the kernel and a set of trusted utilities that control and maintain the kernel audit components and the audit trails.

15.11.1.1 Audit Device Driver

The system implements the kernel audit mechanism as a character special device driver with the normal device interfaces and additional internal (to the kernel) entry points. There are two minor devices (the audit read and audit write device) associated with the audit major device, each of which allows trusted processes access to the kernel audit mechanism.

The kernel audit mechanism defines a control interface, a read interface, and two write interfaces. The control interface (**ioctl( )** calls through the audit write device, which requires the **configaudit** effective privilege) sets the parameters of the audit subsystem. The write interfaces allow the kernel (through an internal interface) and trusted processes (**write( )** to the audit write device, which requires **writeaudit** effective privilege) to append records to the buffered audit trail maintained by the kernel. The kernel buffers these records until a trusted process called the audit daemon (see Section 15.11.1.2) reads them through the read interface and appends them, in compacted form (optionally), to the compacted audit trail.

15.11.1.2 Audit Daemon

The audit daemon receives audit data through the read interface provided by the kernel audit mechanism (through **read**( ) from the audit read device). It is responsible for compacting the data received and appending it to the compacted audit trail. The compacted audit trail is the ultimate destination of audit data. The reduction program reads audit data for the session to be reduced from the compacted audit trail. The audit daemon is involved in the system's shutdown logic to ensure that all audit data generated by the system until system shutdown is appended to the compacted audit trail.

15.11.1.3 Reduction Program

The audit reduction program (**/tcb/bin/reduce**) transforms the binary compressed form of the audit trail to a human-readable format. It uses a selection file to filter the audit records that are to be reported, and converts those records to their printed format. The reduction program re-creates the state of the process that produced the audit record, identifying the user accountable for the audit event and resolving any relative pathnames that identify the objects accessed.

15.11.1.4 ISSO Interface

The ISSO maintains the audit subsystem through the audit-related functions of the ISSO interface program (**/tcb/bin/issoif** for ASCII, or **/tcb/bin/XIsso** for the X interface). This program uses the control interface of the audit subsystem to make dynamic changes to the audit subsystem and manipulates the control files that control the actions of the audit daemon, reduction program, and audit subsystem for subsequent sessions. This program also provides interfaces to back up, restore, and remove audit data associated with named sessions.

15.11.1.5 Trusted Application Support

A library of routines is provided that allows trusted processes to append records to the audit trail. This library uses the write interface to the kernel audit subsystem to append records to the audit trail.

15.11.1.6 Data Structures

The primary kernel audit subsystem data structures are described in the following list:

audit_info structure
: A data structure that stores audit information for the current system call associated with each thread. This structure is reinitialized at the beginning of each call. State information that applies to the process as a whole (for example, selection masks) is stored in the per-process **security_info** structure, and data relevant to the thread context is stored in the **audit_info** structure.

audit_control structure
: The **audit_control** structure contains the global state maintenance variables the kernel audit subsystem uses to control the audit subsystem. The resources the audit subsystem controls are

- The kernel audit buffer
- Selectivity criteria for audit record production
- Buffers for the production of audit records, including pathname buffers

The **audit_control** structure is initialized at the time audit is enabled. It is statically allocated from kernel data space, referenced by the **aud_cont** structure name.

Audit subsystem call table
: The audit subsystem call table stores audit-relevant information about each system call in the underlying system.

The table includes

- Audit event disposition flags
- The default event type
- The default record type

The audit event disposition flags include flags for

- Mandatory audit events
- Audits on error only
- Audits if an event is selected

The table has as many entries as the underlying **sysent** table in the kernel.

15.11.2 Audit Data Flow

This section gives a broad overview of the flow of audit records from their source (trusted processes and the kernel) to their destination (the compacted audit trail). It describes the buffering mechanisms used to make sure that the producers of audit data do not overrun the consumers, and introduces terms that identify the data structures and control mechanisms described in further detail in later parts of this chapter.

Audit records are produced internally in the kernel to record the actions of system calls, and by trusted processes through the audit write device. The kernel audit subsystem must decide whether the record generated should be appended to the audit trail, basing its decision on the selectivity criteria set up by the administrator. If the audit record meets the criteria, it is buffered by the kernel until it can be delivered to the audit daemon.

15.11.2.1 Internal Kernel Buffering

The kernel audit subsystem allocates an audit buffer at the time auditing is enabled. The size of the buffer is an audit subsystem configuration parameter. There is a tradeoff between having a large audit buffer (which increases performance, especially on mainframe configurations) and a small

audit buffer (which reduces the risk of lost data in the event of a system crash).

The subsystem uses this buffer as a staging area for audit record delivery to the audit daemon. The audit device write routine and the audit record generation routine allocate space for the next record in the buffer, causing the calling process to wait if the buffer is full.

The audit daemon requests the next buffer of audit records through the audit read device. If the amount of data accumulated in the buffer exceeds the **read_count** field in the **audit_control** structure, the daemon context is awakened to gather the next buffer of audit data. The audit daemon reads the device interface and receives the next **read_count** bytes of audit data starting at the **read_offset** offset in the kernel buffer. If enough data has been collected, **read_count** bytes are returned to the user context, and pointers into the buffer are adjusted. If not enough data has been collected, the daemon is put to sleep waiting for **read_count** to be exceeded.

Appropriate locks surround manipulation of the audit buffer to maintain the consistency of the data and the control structures describing the buffers.

The source of the data for audit records is the buffer argument to the **write()** system call (trusted process records) and process state information, including system call arguments (kernel audit records). The system call arguments that reside in user space (pathnames and so on) are collected into kernel buffers during the course of the call before being moved into the audit buffer. This avoids having the audit buffer locked while the kernel pages in a pathname from the user process context. Audit records that are written by trusted processes are copied directly into the kernel buffer to avoid multiple copy operations.

15.11.2.2 Compaction Files

The audit daemon appends audit buffers to the compacted[6] audit trail. The compacted audit trail is composed of a sequence of compacted audit output

6. There is an option to produce uncompacted audit output files, but that option is rarely used. The output files will therefore be referred to as compacted, even though it might not be true of all subsystem configurations.

files, each of which is allowed to grow to an administrator-specified size. The daemon switches to a new compacted output file when the current file reaches the specified size.

One configuration parameter specifies a directory that the audit subsystem uses while the system is in single-user mode. Since no other file systems are mounted, audit output files must be placed in a directory on the root file system until the remaining file systems are mounted.

When the system transitions to multi-user mode, the audit daemon is notified to begin creating files in a list of directories that the ISSO specifies as part of subsystem configuration. The daemon closes the output file it had been using on the root file system and opens a new compacted output file in the first directory in the list.

When the daemon encounters a write error to the compacted output file, or free space in the file system that contains that directory falls below a certain percentage of the total file system space, the daemon closes the current output file and opens a new compacted output file in the next directory named in the list.

When the daemon can no longer write in the last directory on the list, it either terminates auditing or shuts down the system (according to ISSO configuration parameters).

The reduction program processes the audit trail by reading sequentially through all compacted audit output files associated with the session being examined. The audit daemon maintains the list of files associated with the audit trail in a session log file, which stores parameters associated with the session and the sequence of compacted audit output files storing session audit records.

15.11.3 Audit Record Formats

Each system call is categorized by an *event type*, which the administrator can use to reduce the amount of data collected (by the audit subsystem) or displayed (by the reduction program **reduce( )**). The kernel determines the event type based on the results of the system call. Trusted processes specify

the event type in the audit records they generate. The event types defined by the system are as follows:

- Startup/shutdown
- Login events
- Process create/delete
- Make available
- Map to subject
- Modify object
- Object unavailable
- Create object
- Delete object
- Change modes
- Access denied
- System administrator actions
- Insufficient privilege
- Resource denial
- Interprocess communication
- Change process control fields
- Audit subsystem events
- Special subsystem events
- Use of privilege event
- Use of authorization event
- Set security level events

The structure of each audit record is defined by one of several *record types*. The record type used for a specific audit record depends on the system call or application event that is being audited.

All records are preceded by a common audit header, which stores the total record length, a timestamp (applied by the source of the audit record), a sequence identifier (applied by the kernel audit subsystem), event and record types, an object type, and a process ID. The process ID is assigned

by the kernel for all events except the special subsystem events type, which may have been produced by server programs or the audit delivery helper program on behalf of another process. The object type identifies the object modified by the operation audited by this record.

15.11.4 Audit Control Flow

This section describes the flow of control through the tasks that the kernel audit subsystem accomplishes in its audit record processing operations.

15.11.4.1 Kernel Audit Record Generation

The collection of system call audit records centers around two hooks in the system call trap handler routine in the kernel. Before the system calls through the **sysent** table to invoke a system call, it calls the **audit_setup()** routine with the call's **sysent** table offset. This routine initializes the **audit_info** fields in preparation for the audit activity of the call. The basic logic for **audit_setup()** is

```
if (auditing is not enabled) {
      remember not to audit this system call
} else {
      initialize a structure of info about the system call to zeroes.
      Look at a table entry for the system call to figure out what
      type of audit record (if any) it should generate, and remember
      this for later.
}
```

When it returns to **syscall()**, the kernel gets the arguments for the system call, then uses the system call number passed in by the user to get the address of the kernel routine that handles the code for that system call.

The second hook is the audit stub routine responsible for collecting system call-specific information necessary for generating the audit record. After the system call has executed, the **audit()** routine is called to decide whether an audit record needs to be produced for that call. The **audit()** routine considers all of the selection criteria that the ISSO has specified and places the information stored during the course of the call into the audit record.

The flow is as follows. The kernel calls the routine for the system call in question. That call is responsible for collecting any information needed to generate the audit record. The information is collected in an **audit_info** structure. Typically, the code for a system call will call appropriate audit stub routines to collect this information.

Eventually, the system call either succeeds or fails, and returns to **syscall( )**, where there is a call to the **AUDIT_GENERATE_RECORD( )** macro. If auditing is not enabled, this does nothing. If auditing is enabled, it calls **audit( )**. This routine collects all the information generated by the previous calls to the stub routines, and decides whether an audit record actually needs to be generated for this system call. If not, it just ignores the information previously collected (if any). The system might decide not to audit because one or more of the following is true:

- Auditing is not enabled.
- Auditing is enabled, but this process is not eligible for auditing according to the audit configuration parameters.
- This process is exempt from auditing because it has the **suspendaudit** privilege or equivalent exemption.
- Auditing is enabled, but this system call does not need to be audited.
- Auditing is enabled, this system call is normally audited, but the arguments the user passed to it were invalid.

If **audit( )** decides that the event must be audited, it goes through a set of routines that format the raw data into a well-defined format and append the formatted data to a buffer in kernel memory. If there is enough data there from the combination of previously written audit data and newly added data, the kernel wakes up **/tcb/bin/auditd**.

The kernel returns back to **AUDIT_GENERATE_RECORD( )** in **syscall( )** from **audit()**, and eventually back to the user program.

At some point, the kernel decides that **auditd** should run (assuming that the kernel previously decided that there was enough audit data collected to make it worthwhile to wake it up). The **auditd** routine reads from the device **/dev/auditr**, which causes the audit data in the kernel buffer to be copied into **auditd**'s memory. The **auditd** routine manipulates the data some more (for example, to compress it) and writes the resulting data into **/tcb/files/audit/CA*** files.

(At some later point, the system manager can look at the generated audit files, using **/tcb/bin/reduce** to read, filter, and format the **/tcb/files/audit/CA*** files.)

15.11.4.2 Pathname Processing

There are at most two pathnames specified as arguments to system calls. Each pathname traversal may cause the traversal of one or more symbolic links. For auditing purposes, both the pathname specified and the directories actually traversed may be useful to the ISSO at reduction time. Therefore, the audit subsystem modifies the pathname processing logic to allow collection of this information during pathname traversal.

The **AUDFLAG_PATH1** and **AUDFLAG_PATH2** flags tell whether the first or second pathname is being traversed. Initially, either **AUDFLAG_PATH1** is set, **AUDFLAG_PATH2** is set instead, or both are cleared, depending on whether the impending traversal is for another pathname or a pathname that is not of interest (for example, if it is the second traversal of the same pathname). After the pathname is successfully copied into the kernel, it is saved into a dynamically allocated buffer, a pointer to which is placed in the thread's **audit_info** structure in the appropriate **si_path** slot; the length is in **si_pathlen**.

When a symbolic link is encountered, a hook in the pathname traversal logic passes the first character in the pathname being translated, the first character following the symbolic link component name, the number of unprocessed characters, and the number of characters remaining in the pathname to a routine that collects the actual path traversed. A new symbolic link pathname buffer is allocated.

If the symbolic link contents start with / (slash), the existing contents of the symbolic link pathname buffer are discarded because the pathname traversal begins again at the **root** directory. Otherwise, the symbolic link pathname is relative, and the pathname traversed so far, the link contents, and the remainder of the pathname are placed in the pathname buffer. The intent of the algorithm is to insert the expanded symbolic link name into the point in the traversal where the symbolic link occurred. This can occur multiple times if more than one symbolic link is encountered during a pathname translation.

Special processing is required in the pathname translation routine if the symbolic link pathname overwrites the pathname buffer before it can be saved in the symbolic link audit routine.

15.12 File System Security Extensions

Security-related file system modifications include

- Changes to the mount table
- Changes to the vnode
- Changes to the file system superblock
- Changes to the in-core and on-disk inodes
- Creation of a security attribute data structure that communicates attributes between the file-system-independent and file-system-dependent layers

15.12.1 Mount Table Security Extensions

A security tag pool must be associated with any file system that is mounted on a system configured for security. Usually, the file system is in extended format, and therefore has tags already associated with it. For backward compatibility, it is also possible to mount unextended format file systems. Since unextended file systems have no space for tag pools, options to the **mount** command must be used in order to specify a set of global tags that apply to all inodes on that file system. These global tags are copied into the **mount** structure for that file system. A flag in the **mount** structure indicates whether or not the file system is in extended format. It is invalid to use **mount** command options to override the tags of an extended format file system, or to attempt a mount of an unextended file system without specifying global tags.

15.12.2 Vnode Security Extensions

The file system architecture provides a clean interface between file-system-independent (vnode) and file-system-dependent (inode) layers. The vnode, as a file-system-independent file header, is extended with an operations vector containing pointers to file-system-dependent security operation routines that implement security functions on the object represented by the vnode. Each file system implementation defines a set of routines that implement these functions. The security attributes are communicated between the file-system-independent (vnode) layer and the file-system-dependent (UNIX file system, System V file system) layer through a virtual security attributes structure that is independent of file system format.

The file-system-dependent routines translate the file-system-independent attribute descriptions into their specific implementation for that file system type.

An additional flag has been defined for the **vnode** structure to indicate that the file is a multilevel parent directory. In addition, a member has been added to the **vnode** structure that references a vector of generic security operations applicable to the file system. Particular file system implementations fill in the vector with the appropriate file-system-dependent routines.

15.12.3 Vnode Security Attributes

A file's security attributes are not stored in the **vnode** structure. Instead, these attributes are stored in the new **vsecattr** structure:

```
struct vsecattr {
   u_short        vsa_valid;         /* which fields are valid (see below) */
   u_char         vsa_policy;        /* policy index for vsa_tag */
   u_char         vsa_tagnum;        /* policy-relative tag index */
   struct vnode  *vsa_parent;        /* parent vnode for tag changes */
   tag_t          vsa_tags[SEC_TAG_COUNT]; /* tag pool */
   privvec_t      vsa_gpriv;         /* granted privileges */
   privvec_t      vsa_ppriv;         /* potential privileges */
   u_long         vsa_type_flags;    /* type flags (MLD, and so on) */
```

```
        uid_t           vsa_uid;        /* POSIX ACL uid(result of tag change)*/
        gid_t           vsa_gid;        /* POSIX ACL group ID (ditto) */
        mode_t          vsa_mode;       /* POSIX ACL mode (ditto) */

};
#define vsa_tag     vsa_tags[0]
```

The **vsecattr** structure is a file-system-independent structure passed between the file-system-independent and file-system-dependent layers through the **getsecattr**() and **setsecattr**() functions, similar to the way the **vattr** structure is passed through the **vnodeops** functions. The **vsa_valid** field is a mask of flags that specify which structure members are to be retrieved from or associated with the file. The flags increase performance by only specifying changed attributes or retrieving desired attributes. File-system-independent routines can retrieve the potential or granted privileges without information about an object's policy tags.

The **vsa_valid** flag can be one of the following values:

VSA_TAG A specific tag pool slot specified by the tag pool offset **vsa_tagnum**

VSA_GPRIV The file's granted privilege set

VSA_PPRIV The file's potential privilege set

VSA_TYPE_FLAGS
The file's type flags, such as whether it is a multilevel parent or child directory

VSA_ALLTAGS
The entire tag pool

The remainder of the fields are storage places for the policy index for the specified policy's tag (if one tag is requested or specified, the first slot in the **vsa_tags** array is used), the policy-relative tag number, the parent vnode, the tag pool, the granted and potential privileges, the type flag, and some information for POSIX ACLs.

15.12.4 Vnode Security Routines

The function pointers in the **vnsecops** structure direct the calling routine to the appropriate file-system-dependent routine. For example, for the UNIX File System (UFS), the **vnsecops** fields contain the addresses of **ufs_getsecattr**( ), **ufs_setsecattr**( ), and **ufs_dirempty**( ). The following macros call through the corresponding **vnsecops** function pointer:

VOP_GETSECATTR( )
: Retrieves selected security attributes from the file specified. No access checking is performed; that is assumed to have been done by the caller.

VOP_SETSECATTR( )
: Changes the specified attributes to the corresponding arguments in the **vsecattr** structure.

VOP_DIREMPTY( )
: Tests whether a directory contains any entries.

A **vsecattr** structure is used as a communication point between these file-system-independent operations and the file-system-dependent routines that manipulate an inode's security attributes directly. **VOP_GETSECATTR**( ) and **VOP_SETSECATTR**( ) are called with arguments specifying the vnode, a pointer to the **vsecattr** structure associated with the vnode, and the calling process's UNIX credentials. **VOP_DIREMPTY**( ) takes a pointer to the directory to be tested, to the parent of the tested directory, and to the the caller's credentials. All three also take a pointer to a place to return an error code.

15.12.5 Superblock Modifications (UFS File System Type)

Extended format file systems have a different magic number than the one used by unextended format file systems. The new magic number is necessary for keeping old programs (written for traditional UNIX systems) from mistakenly believing they know the format of an extended format file system. Without this change, such programs could corrupt extended format file systems. All OSF/1 programs that look at file system magic numbers

have been modified to deal appropriately with both extended and unextended file system formats. The **FsSEC()** macro determines if the superblock's magic number is this new number.

15.12.6 On-Disk Inode Extensions (UFS File System Type)

The on-disk inode for the UFS is described by the **dinode** structure. The security modifications to the on-disk inode consist of a set of security extensions added to the normal **dinode** structure, so that the on-disk inode is now defined by the **sec_dinode** structure.

```
struct sec_dinode {
      struct dinode           di_node;
      struct dinode_sec       di_sec;
};
```

The **di_sec** structure contains granted and potential privilege vectors, the tag pool for the inode, the parent inode number for multilevel directories, and a file type flag. The size of the privilege vectors is implementation-dependent.

```
struct dinode_sec {
      priv_t  di_gpriv[2];     /* granted privilege vector */
      priv_t  di_ppriv[2];     /* potential privilege vector */
      tag_t   di_tag[SEC_TAG_COUNT];  /* security policy tags */
      ino_t   di_parent;       /* inode number of parent of MLD child */
      u_short di_type_flags;   /* type flags (MLD, and so on) */
```

The **dinode_sec** structure also contains the tag pool for the on-disk inode, the parent inode number for multilevel directories, and a file type flag.

15.12.7 In-Core Inode Extensions (UFS File System Type)

The in-core inode for the UNIX File System is described by the **inode** structure (in **ufs/inode.h**). The in-core inode contains all the information of the on-disk inode, plus additional information needed while the file is being referenced by processes. The **inode** structure includes an added **i_disec** member, of the same type as the structure added to the on-disk inode.

```
struct  dinode_sec i_disec;     /* security extension */
```

Since the in-core inode security extensions are only accessed by the kernel, changes to the security extensions do not require recompilation for user-level programs.

15.13 STREAMS Security Extensions

The STREAMS mechanism provides a generalized architecture for implementing communications protocols. The cornerstone of this architecture is the notion of a stream as a sequence of self-contained modules connecting a program at one end (the stream head) to a device driver at the other. The stream provides a full duplex communication path in which each module accepts a message from one of its two neighbors, performs some kind of processing on the message, and then passes it on to its other neighbor. The architecture defines both the internal interfaces and procedures that modules use to communicate with their neighbors in a stream, and the external interfaces that allow programs to create and manage streams and to transfer data.

This section describes the OSF/1 design for trust enhancements to the STREAMS architecture and programming interface. The primary extensions in the OSF/1 STREAMS trust enhancement design are

- The architecture and internal interfaces were extended to associate security attributes with each message that traverses a stream, and to define the attribute format used at the interface between the stream head and its downstream neighbor.
- The STREAMS programming interface was extended to allow programs to obtain the security attributes associated with received messages, to allow trusted applications to specify the attributes to be attached to the messages they send, and to define the attribute format used at the programming interface.
- Hooks were added for auditing data transfers that result from STREAMS **ioctl** commands.
- Binary compatibility was maintained with existing programs that use the STREAMS programming interface.

STREAMS provides a framework for the implementation of a variety of interprocess communication services; however, there are security-relevant issues that cannot be fully addressed by changes to the framework itself, but which must be handled by the specific modules and drivers that work within it. For example, different network protocol suites may encode security attributes in dissimilar ways, making the enforcement of security policies at the time of a process rendezvous a protocol-dependent function. The security extensions ensure that all the necessary information is available to modules and drivers that must perform the security checks.

15.13.1 Local Access Control

One of the ways that a process gains access to a stream is by using the **open( )** system call on a character special file associated with a STREAMS device driver. If the minor device is not already open, a new stream is created and is initially private to the calling process. Otherwise, the returned file descriptor refers to the existing stream, which is shared by all other processes that have opened the same minor device. As with any **open( )** system call, access control is performed based on the attributes (owner, group, mode, and security policy tags) of the device's inode.

On systems configured with MAC, this means that if the device is opened for read/write (as would be typical for a device that implements a communications protocol), the inode and the process must have the same sensitivity level. Even if the open is not for read/write, the **ioctl** system call performs another access check to ensure that the process and inode have the same level.

Together, these facts mean that, for all practical purposes, a STREAMS device cannot be used concurrently by untrusted processes at different sensitivity levels unless its inode has a WILDCARD label. For devices that implement network protocols, it is usually essential that they be equally accessible to processes at all levels. If such a driver allows multiple concurrent opens of the same minor device, then processes at different levels could effectively bypass the system's MAC policy and communicate with each other by pushing and popping modules and making other changes to the configuration of the stream. To guard against this, drivers that want to provide multilevel service must either prohibit concurrent opens of the same minor device, or perform their own MAC enforcement. Drivers that can only be opened indirectly through the **clone** device automatically avoid the

problem, since every **open( )** call results in the creation of a new stream that is private to the opening process.

A driver that wants to perform its own access control should associate a tag pool with each minor device. Then, upon the first **open()** call on a minor device, it should initialize the tag pool by calling **SP_OBJECT_CREATE( )**, and on each subsequent **open()** call, it should perform an access check by calling **SP_ACCESS( )**. The result is that when a new stream is created, it inherits a sensitivity label from the creating process and, for the remainder of its existence, is only accessible to trusted processes or processes at the same level as the creator. This is the same general approach used by the OSF/1 trust-enhanced pseudo-tty driver.

15.13.2 Internal Interfaces

Internally, each message that passes through the stream is augmented with a complete set of security attributes, which consists of values specified by the caller combined with default values taken from the current state of the calling process. This set of attributes is attached to the first message block in each message. Several functions that perform processing at the stream head are modified to copy attributes from user space and translate them into their internal format, and vice versa. Because the attributes are incorporated into the basic message structure, the standard interfaces between modules remain unchanged.

15.14 Socket Security Extensions

OSF/1 security extensions to sockets support MAC and allow a trusted server process to receive the security attributes of clients with client requests. The socket extensions have been implemented only on UNIX domain sockets, not on Internet domain sockets.

15.14.1 Socket Data Structures

Each socket data structure has a new **so_tag** member that includes a tag pool in the socket data structure. The allocation of tags in the pool is the same as the layout for other objects in the system.

The socket mechanism includes the concept of **rights**, which is a list of file descriptors that can be passed in a message across a socket connection. The system uses a data structure, called the **rights buffer**, to contain those file descriptors in transit and to contain the privileges of the sending process. The buffer is structured as a set of descriptors, each of which contains a file descriptor and privileges. Each descriptor is formatted as a rights type followed by a length. The contents of the descriptors are as follows:

SEC_RIGHTS_FDS
The file pointers of the file descriptors passed

SEC_RIGHTS_PRIVS
The effective privileges of the process

There is an additional socket option, **SO_EXPANDED_RIGHTS**, which a process can set if it is interested in receiving the privilege mask on each **recvmsg**( ) call. The user process rights buffer received has the same format as described for the kernel.

15.14.2 Socket Control Flow

A process creates a socket (or two sockets) using the **socket**( ) (or **socketpair**( )) system call. At socket creation time, the system initializes the socket's tags through a call to **SP_OBJECT_CREATE**( ). When the server process calls **listen**( ) to create a queue for incoming connections, the socket becomes the prototype socket for new connections to the server.

When a client process calls **connect**( ), specifying a UNIX domain socket (the connecting socket) and a pathname, the system checks write access to the socket file named by the pathname and creates a new socket, which is added to the prototype socket's pending connections queue. The tag pool for the new socket is copied from the connecting socket's tag pool.

When a server process calls **accept**( ) to complete the connection protocol, the system calls **SP_ACCESS**( ) to check **SP_IOCTLACC** access between the connecting socket and the prototype socket (which is implemented as an

equality check for MAC and no check for DAC). If the check succeeds, the new socket is removed from the pending connections queue, and a new file descriptor is allocated and returned to the server. Appropriate data structure references are made between the connecting socket and the new socket to establish the connection.

If the client sends a message without specifying a rights buffer, the system creates one **SEC_RIGHTS_PRIVS** descriptor containing the effective privilege vector of the sending process. The process may specify an alternate privilege descriptor if the privilege vector specified is a subset of the union of the process's base privilege set and the current program's potential privilege set. The process may pass file descriptors if it has both the **allowmacaccess** and **allowdacaccess** effective privileges.

15.15 Loader Security

In either unextended or secure OSF/1, a privileged program (**setuid**, **setgid**, or executing with any privilege bits set in its privilege vector) will always be loaded through the default loader. This ensures that users cannot use their own loaders to load arbitrary privileged programs.

The loader design also protects the user of shared libraries. The way the loader finds its shared libraries is by consulting the installed package tables (see Chapter 8). The global installed package table is secured by virtue of its being stored in a file in a known-secure part of the file system, which is writable only by **root**. This ensures that any libraries found through the global installed package table are secure.

The private installed package table is inherited across **exec( )** by the fact that it is stored in a memory segment allocated with the **keep-on-exec** bit turned on; this causes it to be retained in the process's address space across the **exec( )** call. The loader finds it during loader initialization (as part of the inheritance operation) by making the **getaddressconf( )** call to find the address to look at, and the **mvalid( )** call to verify that there is something there.

The **exec( )** call will not retain any segments, even segments marked **keep-on-exec**, in the address space when the program it is executing is privileged. This check is implemented in the **vm_map( )** call, which is passed an *is_priv* flag from **exec( )**. Therefore, when a privileged program is run, its loader will never find a private installed package table in its address space,

and will always use the global installed package table. Thus, it is not possible for a user to **inlib** a library and have that library used in a privileged application.

15.16 Mach Subsystem Security

The basic security issue for the Mach subsystem involves port rights. Task ports can be used to replace the entire contents of a task's address space; in the absence of security checks, users could **exec** a privileged task and dump their own code into it. Mach exceptions transfer rights on the task port to the recipient of the exception message, and therefore, exception ports must be checked as well.

The **port_secure**( ) routine determines whether the given task has the only access rights to the port in question. The **exc_port_secure**( ) routine determines if an exception port is secure. This is the case if either the **ux_exception** task is the receiver, or a privileged task is the receiver. (A privileged task in this context is one whose effective user ID is 0 (zero) in unextended OSF/1, or a task with the **debug** effective privilege in secure OSF/1.) The **task_secure**( ) routine determines whether the given task can be manipulated by tasks other than itself, using the IPC interface. The routines ensure that an **exec** of a **setuid** or otherwise privileged program will fail if someone else is holding a port right they should not have. The **task_secure**( ) check is implemented in terms of **port_secure**( ) and **exc_port_secure**( ) (that is, it calls them on all relevant ports).

The **task_by_unix_pid**( ) routine gets the task port for a task on the same host as another task whose port the calling task already has rights to (called the target task). This can only be done if the specified task has the same user ID as the target task, or if the calling task is privileged (as defined previously).

The **trap_name**( ) routine, which provides the currently executing task/thread with one of its ports, has been extended to protect the privileged host port. Without this check, a user might gain complete control of resources on the host, including the ability to get the task ports.

Of these, only the last two routines are externally callable. The others mentioned are all internal kernel routines.

The Mach interfaces have not been explicitly secured for the OSF/1 security configurations.

15.17 Modified Data Structures

The secured OSF/1 system modifies a number of OSF/1 kernel data structures to implement its security extensions. A number of file system data structures are modified, specifically the following:

Mount table
: The mount table is extended to include a flag bit indicating whether the mounted file system supports additional attributes (is an extended format file system), and a tag pool that stores attributes (sensitivity level, ACL) that apply to the entire file system for traditional format systems.

File structure
: The file-system-independent data structure for a file stores a separate operations vector that points to a set of operations for setting and retrieving attributes on extended format file systems. If the system is configured with MAC, the file-system-independent file data structure also stores a flag to indicate whether the file is a multilevel parent directory.

File system superblock
: The on-disk superblock for extended format file systems stores a magic number that is different from the one traditionally associated with the specified file system type. The nonstandard magic number indicates to unmodified software that the underlying file system cannot be manipulated by software not prepared to deal with the changed format.

On-disk inode
: The on-disk file header stores a tag pool, two privilege vectors, and a flag word. The strategy for incorporating the fields in the on-disk data structures is file-system-dependent.

In-core inode
: The in-core file header stores the same fields as the on-disk inode.

Some other objects have tag pools. Those that are dynamically allocated and not visible to user programs are in the data structure (for example, sockets). Those that are allocated in tables and not visible outside the kernel have tag pools allocated in parallel tables.

15.18 New Data Structures

A number of new data structures have been added to store the additional security data associated with each subject and object in the system. These include the following:

security_info
: Stores per-process, security-relevant state information that spans system calls. The **security_info** structure for the currently executing process is typically accessed with the **SIP** macro.

audit_info
: Stores information gathered during system call execution to support the audit subsystem. The information in this structure is transient, and does not need to be maintained between system calls. In OSF/1, the **audit_info** structures are per-thread. The **audit_info** structure for the currently executing thread is typically accessed with the **AIP** macro.

audit_control
: Stores the internal state of the audit subsystem.

udac_t
: Contains an object's unsecured OSF/1 security attributes. The security policies configured into the system use this structure to coordinate the unsecured OSF/1 DAC decision with other security attributes of the object.

obj_t
: A union of a number of object identifiers, including the file pathname and open file descriptor number, and the process, semaphore, shared memory, and message queue IDs. It is used at the system call interface when security attribute changes to the various unsecured OSF/1 software abstractions are specified.

attr_t
: Used to pass internal representations of security attributes between user and kernel space.

attrtype_t Used to indicate whether an attribute is associated with a subject or an object.

dac_t Contains the user and group IDs, and unsecured OSF/1 permission bits.

Glossary

access control list (ACL)

A variable-length list that is associated with an object (typically a file). The entries in the list identify users or groups of users who may access the object, and what kind of access they have. *See also* **discretionary access control (DAC)**.

address map

A data structure that the kernel uses to manage a task's virtual address space. It is a doubly linked list of address map entries.

address map entry

A data structure that maps a virtual memory object into a task's address space.

address translation

A mechanism that translates a program's logical or virtual addresses to physical memory addresses. Typically address translation is implemented by the hardware's *memory management unit*.

Advanced Japanese EUC (AJEC)

A Japanese implementation of the Extended UNIX Code (EUC) encoding method. AJEC allows for combining ASCII, phonetic Kana, and ideographic Kanji characters. *See also* **Extended UNIX Code (EUC)**.

anonymous memory

A region of virtual address space that contains data generated by the process as it executes. For example, a process's *heap* and *user stack* are regions of anonymous memory.

AST

See **asynchronous system trap (AST)**.

asymmetric copy-on-write

A copy-on-write mechanism that allows one task to retain exclusive write access to permanent data that is shared copy-on-write with other tasks. *See also* **permanent data**, **copy-on-write**.

asynchronous system trap (AST)

A software-initiated event that interrupts a thread's execution as it transitions from kernel mode to user mode. The OSF/1 kernel uses ASTs to implement context switching.

audit

To audit means to record security-relevant events. "Audit information must be selectively kept and protected so that actions affecting security can be traced to the responsible party."[1]

authentication

1. The verification of a principal's network identity.
2. A mechanism that is used by the system to verify that the user is in fact who she or he claims to be. *See also* **identification and authentication (I&A)**.

authorizations

There are two kinds of authorizations: command authorizations and kernel authorizations. Command authorizations control user access to programs or program subfunctions. The system restricts certain activities to certain users by allowing a user to perform an action only if he or she possesses the required authorization. Authorizations are enforced by trusted applications and protected subsystems.

Kernel authorizations, or override authorizations, are associated with specific kernel actions that are allowed to privileged users. These authorizations control the ability of trusted commands to override basic system constraints. They also affect the way that a user can enable privileges for all commands he or she executes and they limit the privileges that a user can associate with a program.

Kernel authorizations are different from command authorizations in that kernel authorizations are used to enable specific security policy overrides in certain trusted applications, while command authorizations signal trusted commands to grant the user the required operational rights. Kernel authorizations are directly related to kernel-recognized

1. *Trusted Computer System Evaluation Criteria (TCSEC)* (CSC-STD-001-83), U.S. Department of Defense, National Computer Security Center, August 15, 1983. Requirement 4. Preface.

privileges; command authorizations grant the right to perform some class of operations, without regard to how those operations are actually implemented, or what privileges are normally required to perform them. Command authorizations allow that right to be granted to any subset of users on the system. *See also* **privileges**.

background process

1. A process that does not require operator intervention but can be run by the computer while the workstation is used to do other work.
2. A mode of program execution in which the shell does not wait for program completion before prompting the user for another command.
3. Contrast with **foreground process**.
4. A process that is allowed to execute as long as it does not attempt to access the terminal. When it attempts to access the terminal, the kernel suspends the process. *See also* **foreground process**.

backing object

The VM object that contains the original data when a task shares data copy-on-write symmetrically. When the task attempts to write the data, the page being written is physically copied and the new page is inserted in a shadow VM object. *See also* **shadow object**.

backing store

1. The collection of off-screen, saved pixels that are maintained by the server.
2. Secondary storage (ususally on a disk drive) that is used to store data from resident memory when the data is paged out or swapped.

bad sector relocation

The sector relocation that is performed by the Logical Volume Manager when it encounters a hard (uncorrectable) bad sector. *See also* **Logical Volume Manager (LVM)**.

block

1. A group of contiguous records or data that is recorded or processed as a unit.
2. In data communications, a group of records that is recorded, processed, or sent as a unit.
3. In programming languages, a compound statement that coincides with the scope of at least one of the declarations that is contained within it. A block may also specify storage allocation or segment programs for other purposes.
4. A group of contiguous records, or data, that is recorded or processed as a unit.
5. When a thread attempts to access a system resource that may not be immediately available, the kernel blocks the thread until the resource becomes available. A thread that is blocked is *sleeping*.

block device

1. A device that is accessed as a set of sequential blocks of data through a block interface. *See also* **character device**.
2. One of the types of files in the file system, which is described by an inode.

bss

In a program that is to be loaded into memory, **bss** is the portion that is to be initialized to some constant, usually 0 (zero). The term is from an old assembler directive, ''block started by symbol.'' *See also* **object file format**, **data section**, **text**.

busy page

When a virtual page is about to become involved in a paging operation, the kernel marks the page's **vm_page** data structure as *busy* to prevent other threads from initiating additional paging operations on the page.

canonical mode

A tty input processing mode where input is collected and processed one line at a time. *See also* **noncanonical mode**.

catch a signal

A process may choose to catch a signal by installing a signal handler routine. When the signal is delivered, the kernel arranges to execute the signal handler routine, and so the signal is caught.

character device

A device that provides either a character-stream-oriented I/O interface or, alternatively, an unstructured (raw) interface. Devices that are not character devices are usually block devices.

child process

1. A process, which is spawned by a parent process, that shares the resources of the parent process.
2. A new process that is created when another process executes.

clearance

The highest sensitivity level available to a user. *See also* **sensitivity label**, **mandatory access control (MAC)**.

client

1. CDS: Any application that interacts with a CDS server through the CDS clerk.
2. DTS: Any application that interacts with a DTS server through the DTS clerk.
3. RPC: The party that initiates a remote procedure call. Some applications act as both an RPC client and an RPC server. *See also* **server**.
4. DFS: A consumer of resources or services. *See also* **server**.

5. GDS: The client consists of an application that links the DUA library, the C-stub that handles the connection over the communications network for accessing a remote server, and the DUA cache.
6. A program that is written specifically for use with the X Window System. Clients create their own windows and know how to resize themselves.
7. The portion of a distributed program that issues requests for service to a server. The client's address space is separate from the server's address space; the two programs may reside on separate machines. *See also* **server**.

client/server

A model of computer interaction in which a server provides resources for other systems on a network, and a client accesses those resources.

cluster

1. Any configuration of interconnected workstations for the purpose of sharing resources (for example, local area networks, host attached workstations, and so on).
2. A group of storage locations allocated at one time.
3. A station that consists of a control unit (cluster controller) and the terminals that are attached to it.
4. *See also* **page cluster**.

code set

1. A collection of characters with assigned code values. For example, ASCII contains a specified group of characters; each character has an assigned value in the set.
2. The set of binary values that is needed to represent all the characters in a language.

COFF

See **Common Object File Format (COFF)**.

Common Object File Format (COFF)

The object file format that is used in System V, Release 3 UNIX environments.

configuration manager

The daemon process that performs configuration at boot time and that handles requests for dynamic changes to the configuration.

configuration method

User-supplied code that provides the configuration manager with a description of how to configure dyanamic subsystems into the system. *See also* **configuration manager**.

context

An environment for computation; for example, virtual memory and CPU state.

context switching

Occurs when a CPU switches from executing one thread to executing another.

copy map

A copy of one or more address map entries that is used to pass data between tasks.

copy object

A virtual memory object that is created during an asymmetric copy-on-write operation when the task that has read/write access to permanent data first attempts to write data that is shared copy-on-write. Before the task can write a page of data, it must push a copy of the page to the copy object so that the other task that is sharing the data retains access to the data as it existed when the two tasks began sharing the data copy-on-write. *See also* **asymmetric copy-on-write**.

copy-on-write

1. An option that creates a mapped file with changes that are saved in the system paging space, instead of saving the changes to the copy of the file on the disk.
2. A mechanism where data can be shared between two or more tasks and copied only when one of the tasks writes the data. In OSF/1, there are two copy-on-write mechanisms: symmetric copy-on-write and asymmetric copy-on-write. *See also* **asymmetric copy-on-write**, **symmetric copy-on-write**.

core file

A file that records the state of a process at the time it was terminated. The file includes the contents of the process's virtual address space. The kernel produces a core file of a process when it delivers certain signals to the process that force the process to terminate.

daemon

1. A program that runs unattended to perform a standard service. Some daemons are triggered automatically to perform their task; others operate periodically. An example is the **cron** daemon, which periodically performs the tasks that are listed in the **crontab** file. Many standard dictionaries accept the spelling **demon**.
2. A process or thread that performs system-related operations. Generally, daemons are started during system initialization. Daemons usually sleep when their services are not needed. The pageout daemon is an example of a daemon in OSF/1.

data section

The portion of an object file or process address space that contains initialized and uninitialized data.

deadlock

1. An error condition in which processing cannot continue because each of two elements of the process is waiting for an action by or a response from the other.

2. An unresolved contention for the use of a resource.
3. An impasse that occurs when multiple processes are waiting for the availability of a resource that does not become available because it is being held by another process that is in a similar wait state.
4. In multithreaded programming, the condition that is caused when one or more threads block indefinitely, each waiting for the other to give up the specified lock.

demand paging

A memory management policy where text and data is brought into resident memory only when it is referenced.

dirty

A page that has been modified while in resident memory. The memory management system must save all dirty pages in secondary storage before reusing their resources in resident memory.

discretionary access control (DAC)

The "Orange Book" defines discretionary access control as "A means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject (unless restrained by mandatory access control)."[2] *See also* **access control list (ACL), mandatory access control (MAC), security policy**.

distributed file system

A file system that is composed of files or directories that physically reside on more than one computer in a communications network.

---

2. *Trusted Computer System Evaluation Criteria (TCSEC)* (CSC-STD-001-83), U.S. Department of Defense, National Computer Security Center, August 15, 1983. Glossary.

distributed program

A program that is distributed among multiple tasks. The tasks may run on different machines. A client/server application is an example of a distributed program.

domain

A type of addressing that is used by a network layer. For example, the Internet protocol family (IP, TCP, UDP) comprises the Internet domain.

dominance

The method that is used to compare sensitivity levels. Level **A** is said to dominate level **B** if **A**'s classification is greater than or equal to **B**'s (according to numeric value of the classification) and if **A**'s compartments are a superset of **B**'s. Two sensitivity labels are equal if their classifications and compartment sets are the same. If **A** dominates **B**, and if they are not equal, **A** is said to *strictly dominate* **B**. *See also* **sensitivity label**, **mandatory access control (MAC)**.

dynamic configuration

A means of configuring a subsystem that involves loading it into an executing kernel. There is also dynamic unconfiguration.

Extended UNIX Code (EUC)

A character encoding scheme that allows a combination of several code sets to be used simultaneously. It can be used as an encoding method for code sets that are composed of single or multiple bytes.

fictitious page

A **vm_page** data structure that does not point to an actual page frame. Fictitious pages are used by the memory management system to represent pages that are involved in paging operations.

file descriptor

1. A small positive integer that the system uses instead of the filename to identify the file.
2. A small unsigned integer that a UNIX system uses to identify an open file. A process creates a file descriptor by issuing an exist when it is no longer held by any process.

file-based privileges

Privileges in which effective user identity normally does not determine privilege. Rather, trusted programs use the privilege library to enable and disable privileges around all operations that require them. *See also* **privileges**, **privilege bracketing**.

foreground process

1. A process that must run to completion before another command is issued to the shell.
2. A process that has access to the controlling terminal. The command interpreter waits for the current foreground process to finish executing before prompting the user to enter another command.

framework

A set of interfaces and associated code that provides subsystems access to the system's resources. Example frameworks include STREAMS, sockets, and the virtual file system.

Green Book

Officially entitled *U.S. Department of Defense Password Management Guideline*, this book offers criteria for identification and authentication management. It is called the "Green Book" because of the color of its cover as part of the Rainbow Series. *See also* **Orange Book**.

group ID (GID)

See **group number (GID)**.

group number (GID)

A unique number that is assigned to a group of related users. The group number can often be substituted in commands that take a group name as an argument.

hashing

1. A method of transforming a search key into an address for the purpose of storing and retrieving items of data.
2. Encoding a character string as a fixed-length bit string for comparison. The encoding may not necessarily be unique.

heap

1. A collection of dynamically allocated variables.
2. The region of a process's virtual address space that provides storage for global data. *See also* **bss**.

hook

The act of configuring a dynamic subsystem into the kernel.

identification and authentication (I&A)

Identification is how a user tells the system who she or he is. Authentication is how the system verifies that the user is in fact who she or he claims to be. In OSF/1, the authentication subsystem does more than provide password management. It is a framework where processes, trusted applications, and the kernel work together to ensure the identity of users and their processes.

information systems security officer (ISSO)

The ISSO is an administrative role that sets system defaults for users, maintains security-related authentication profile parameters, modifies non-ISSO user accounts, administers the audit subsystem, assigns devices, and ensures system integrity.

interrupt service routine (ISR)

A routine that executes as a direct result of an event, such as a device or timer interrupt. It is an interrupt handler.

job control

The facilities for monitoring and accessing background processes.

kernel map

In OSF/1, the address map data structure that describes the kernel's address space.

kernel mode

1. The state in which a process executes kernel code. Contrast with **user mode**.
2. A privileged mode of execution in which the CPU can execute kernel code and access kernel data structures. A user process executes in kernel mode when it *traps* into the kernel by executing a *system call*. *See also* **system call**.

kernel stack

The stack that is used by the CPU when a process executes in kernel mode. *See also* **user stack**.

kernel task

In OSF/1, the task that is associated with the kernel's virtual address space.

kernel thread

A thread that executes within the kernel task to perform system-related operations. For example, the pageout daemon executes as a kernel thread.

large sparsely filled address space

An address space whose regions of allocated memory are separated by large regions of unallocated memory.

lazy evaluation

A programming optimization that defers performing an operation until it absolutely must be performed. Copy-on-write is an example of an operation that is lazily evaluated.

least privilege

The "Orange Book" requirement that stipulates that users and programs possess the least number of privileges possible to perform operations. *See also* **privilege bracketing**, **Orange Book**.

line discipline

1. The asynchronous communications user interface for a tty, which includes the POSIX and the Berkeley line disciplines.
2. A software module that provides an asynchronous communications user interface for a tty. The line discipline performs the input and output processing for ttys and ptys, as specified by the **termios** structure.

load average

The measure of the load on the system's CPUs. In OSF/1, it is calculated as the number of runnable threads divided by the number of CPUs averaged exponentially over time.

loader switch

A data structure of the OSF/1 loader that provides, for each of several object formats, a set of entry points defining a format-dependent manager that is appropriate to that format.

locale

1. The language. geographic location, and software environment that is required to support the local language and customs. For example, the environment required to support the French language in Canada is a locale. A locale can include information about the language, the code set that is used to represent the language, the collating sequence, and cultural requirements for printing numeric and date values.
2. The international environment of an application program that defines the language-dependent behavior of the program at run time. An application derives the locale based on internal procedures and a set of implementation-defined values.

logical extent

The unit of allocation of logical volume space. Each logical volume consists of a number of logical extents. All extents within a given volume group are of the same size. *See also* **Logical Volume Manager (LVM), physical extent**.

logical track group (LTG)

Each logical track group consists of 32 consecutive pages. The Logical Volume Manager uses the logical track group internally to ensure mirror consistency, and to perform mirror resynchronization. *See also* **Logical Volume Manager (LVM), mirrored**.

logical volume

1. A direct access storage device (DASD) that is composed of a collection of physical partitions that are organized into logical partitions, all contained in a single volume group. Logical volumes are expandable and can span several physical volumes in a volume group.
2. Logically contiguous areas of disk.
3. A volume that is implemented by the LVM. To users and file systems, logical volumes appear as devices. A logical volume can be thought of as a virtual disk drive, although it may map to multiple physical volumes. *See also* **Logical Volume Manager (LVM), physical volume, volume**.

Logical Volume Manager (LVM)

An OSF/1 subsystem that provides a level of abstraction between physical volumes and the file management subsystem that allows a file system, or even a single file, to span multiple physical volumes.

LVM

See **Logical Volume Manager (LVM)**.

Mach Interprocess Communication Subsystem (Mach IPC)

A Mach kernel subsystem that provides primitives and operations that allow tasks to send messages to one another.

main memory

See **resident memory**.

mandatory access control (MAC)

Defined in the "Orange Book" as "A means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorization (that is, clearance) of subjects to access information of such sensitivity."[3] *See also* **discretionary access control (DAC), security policy**.

mbuf

A data structure that describes a block of data. Mbufs are used by some communication subsystems.

memory management unit (MMU)

A hardware component that performs address translation and implements the hardware's memory protection scheme. *See also* **address translation**.

memory manager

A task that manages paging operations on memory objects by using the external memory manger interface. A memory manager may run in user space and implement application-specific memory objects.

3. *Trusted Computer System Evaluation Criteria (TCSEC)* (CSC-STD-001-83), U.S. Department of Defense, National Computer Security Center, August 15, 1983. Glossary.

memory mapped address space

An address space that contains logical addresses. An executing process references its instructions and data with logical (or virtual) addresses, which the memory management unit translates to physical addresses. *See also* **virtual address space**.

memory object

An object that represents a set of virtual pages that reside in secondary storage. Each memory object is managed by a memory manager.

method

1. The subsystem-specific part of the configuration manager that runs in user space.
2. In object-oriented programming, a function that is used to perform operations on an object. The method is part of the object.

mirrored

1. The state of physically replicated data that is stored in a logical block. Mirrored data refers to the copies of data that are stored in physical extents (blocks) that map to a unique logical extent. Data can be singly mirrored (one additional copy) or doubly mirrored (two additional copies).
2. The state of physically replicated data that is stored in a logical block. Mirrored data refers to the copies of data that are stored in physical extents (blocks) that map to a unique logical extent. Data can be singly mirrored (one additional copy) or doubly mirrored (two additional copies).

MMU

See **memory management unit (MMU)**.

mount point

1. DFS: An access point to a fileset in the DFS file tree. If a fileset has been mounted, the resulting mount point looks and acts like a directory in the file tree.
2. The local directory of an NFS client where the remote directory is mounted.
3. A file on which a file system has been attached.

multilevel directory

A directory that has separate child directories for each sensitivity level, and is used to implement directories that must be accessible to processes at more than one sensitivity level. When an unprivileged process references a multilevel directory, it is automatically diverted into the child directory corresponding to the process's sensitivity level. *See also* **sensitivity label**, **mandatory access control (MAC)**.

multilevel secure

Defined in the "Orange Book" as "A class of system containing information with different sensitivities that simultaneously permits access by users with different security clearances and needs-to-know, but prevents users from obtaining access to information for which they lack authorization."[4] *See also* **sensitivity label**, **mandatory access control (MAC)**.

multiprocessor

A computer having more than one central processing unit (CPU). The CPUs generally share a resource such as memory or a bus, allowing some degree of cooperation.

noncanonical mode

A tty input processing mode where input character erase and killing are eliminated, making input characters available to the user program as they are typed. *See also* **canonical mode**.

4. *Trusted Computer System Evaluation Criteria (TCSEC)* (CSC-STD-001-83), U.S. Department of Defense, National Computer Security Center, August 15, 1983. Glossary.

object file format

A specification for the output of an assembler, compiler, or linker; covers the representation of bss, text, and data sections, and their mappings, as well as imported and exported symbols. *See also* **bss, text, data section**.

Orange Book

Officially named the *U.S. Department of Defense Trusted Computer System Evaluation Criteria (TCSEC).* This book is the U.S. Government's definitive guide to the development and evaluation of trusted computer systems. It is referred to as the "Orange Book" because of the color of its cover. It is part of a series of government security books that are called the Rainbow Series. *See also* **Green Book**.

OSF/ROSE

The object file format that is supported by OSF/1 for user programs and for kernel extensions.

out-of-line data

With respect to the Mach IPC subsystem, data is said to be passed out-of-line when the message that is sending the data contains pointers to the data instead of the data. *See also* **Mach Interprocess Communication Subsystem (Mach IPC)**.

package

1. A specified group of related OM classes, denoted by an Object Identifier.
2. In the OSF/1 loader, a collection of object entities that share a common name space. Symbol names are unique within a package. Symbols from different packages may bear identical symbol names because they are distinguished by their package names.

page

1. A block of instructions, data, or both.
2. The number of lines that can fit into a window.
3. In a virtual storage system, a fixed-length block that has a virtual address and is transferred as a unit between real storage and auxiliary storage.

page cluster

A group of adjacent virtual pages.

page fault

1. A program interruption that occurs when a program refers to a page that is not in real memory.
2. A program interruption that occurs when a program attempts to access a page that is either not in resident memory or is resident but is protected against the type of access. For example, if a page is protected against write access and a program attempts to write the page, the attempt generates a page fault.

page fault handler

The part of the kernel that executes when a thread generates a page fault. The page fault handler handles validity faults and protection faults. If it is unable to resolve a page fault, the page fault handler sends a signal to the process that causes the process to be terminated. *See also* **page fault**, **protection fault**, **validity fault**.

page frame

1. In real storage, a storage location having the size of a page.
2. An area of main storage that contains a page.
3. A fixed-length unit within the system's resident address space.

page table

A machine-dependent data structure that is used by the hardware's memory management unit to perform address translation. A page table contains page table entries, each of which maps a virtual page to a physical location in the system's memory heirarchy.

paging

1. The action of transferring instructions, data, or both between real storage and external storage.
2. Moving data between memory and a mass storage device as the data is needed.

parallel processing

The condition in which multiple processors are executing a single image, such as the OSF/1 kernel.

parallel program

A program that performs its operation by using more than one thread of control.

permanent data

A process's data that exists before the process executes and after the process exits. For example, a process's text is permanent data: it resides in a file in secondary storage. *See also* **temporary data**.

physical extent

1. A physical extent is a specific, contiguous region of the disk where the data resides.
2. The unit of allocation of physical volume space. All of the physical extents within a given volume group are the same size. *See also* **Logical Volume Manager (LVM)**, **logical extent**.

physical map (pmap)

A data structure that provides a handle to the machine-dependent representation of a task's virtual-to-physical translations.

physical volume

1. A read/write fixed disk that is physically connected to a computer.
2. A contiguous area of a physical disk. *See also* **volume group**, **logical volume**, **logical volume manager (LVM)**.
3. A contiguous area of a physical disk drive. This can mean either an entire disk or a portion of the disk (for example, a UNIX partition). Usually the LVM uses an entire physical disk as a physical volume, but it allows for using individual UNIX partitions as physical volumes. *See also* **volume**, **logical volume**, **logical volume manager (LVM)**.

PID

See **process ID (PID)**.

pmap

See **physical map (pmap)**.

pmap module

The kernel module that is responsible for managing physical maps. The pmap module contains all machine-dependent code that is associated with the memory management system.

port

1. A part of the system unit or remote controller to which cables for external devices (display stations, terminals, or printers) are attached. The port is an access point for the entry or exit of data.
2. An entrance to or exit from a network.
3. Allows the programming changes that are necessary to permit a program that runs on one type of computer to run on another type of computer.
4. An access point for data input to or data output from a computer system.

preload cache

An operating system resource that contains all globally available shared libraries. These libraries are fully resolved and relocated and are mapped directly into a process's address space when the process is loaded.

privilege bracketing

The "Orange Book" requirement that stipulates that users and programs possess privileges for the shortest time necessary to perform operations. *See also* **least privilege**, **Orange Book**.

privileges

Access rights that are granted to a process. When making operational decisions, the security-extended kernel checks which of approximately 30 defined privileges (depending on the configuration) a process possesses. A privilege set consists of individual privileges from the privileges that the operating system recognizes. Before allowing a privileged operation, the kernel verifies that the privilege is in the process's effective privilege set. The effective privilege set is dynamic. It can change when the process executes a new program and when trusted programs enable and disable privileges. *See also* **file-based privileges**.

process

1. A sequence of actions that is required to produce a desired result.
2. An entity receiving a portion of the processor's time for executing a program.
3. An activity within the system that is started by a command, a shell program, or another process. When a program is running, it is called a *process*.
4. In a computer system, a unique, finite course of events that is defined by its purpose or by its effect, achieved under given conditions.
5. Any operation or combination of operations on data.

6. In the operating system, the current state of a program that is running. This includes a memory image, the program data, the variables that are used, general register values, the status of opened files that are used, and the current directory. Programs that are running in a process must be either operating system programs or user programs.
7. An address space and the threads of control that execute within it, as well as the associated system resources.
8. An address space, the threads of control that execute within it, and the associated system resources.

process code

An encoding scheme that is used when a program is manipulating or processing characters. Process codes are designed for efficiently manipulating character data, but should not be used to communicate character data. Process codes should only be used in a single program execution and should not be written to a file.

process context

See **context**.

process control block (PCB)

A data structure that is associated with a thread that is used to store the thread's hardware state when the thread is not executing. The process control block is a machine-dependent data structure.

process ID (PID)

A unique number that is assigned to a UNIX process.

processor

In OSF/1, a data structure that is used to manage the state of a CPU. Each of the system's CPUs has a *processor* data structure.

processor aging

A mechanism that is used by the scheduler to gradually decrement a process's CPU utilization so that the process's priority rises if it has not executed recently.

processor server

A privileged program that applications can use to create new processor sets and allocate processors to those sets.

processor set

A data structure that is used to manage the state of a group of CPUs. By default, a thread is scheduled to run on the system's default processor set. The kernel ensures that at least one CPU is assigned to the default processor set so that kernel threads always have access to a CPU.

program exception

An interruption in the sequence of a program's instructions that is caused by the current instruction. For example, if the instruction references a page that is not resident, it generates a page fault exception. Program exceptions include system calls, which *trap* the process into the kernel, illegal instructions, and attempts to divide by zero.

protection fault

A page fault that is generated when a thread attempts to reference a page in a way that violates the page's protection. *See also* **page fault**, **page fault handler**, **validity fault**.

pseudoterminal (pty)

1. A special file in the /**dev** directory that effectively functions as a keyboard and display device.
2. A special file in the /**dev** directory that effectively functions as a keyboard and display device. It acts like the tty, except that it connects two user processes (instead of a process and a hardware terminal). *See also* **tty**.

pthreads library

A subroutine libarary that application programmers can use to implement multithreaded programs.

raw mode

See **noncanonical mode**.

region

1. In the OSF/1 loader, a contiguous portion of loadable space, as specified in the object file. It has a starting virtual address, a size, a mapped address, a protection, and (optionally) a name and flags.
2. An area within a bitmap, a pixmap, a screen, or a window.

remote procedure call (RPC)

A procedure call that is executed by an application procedure that is located in a separate address space from the calling code.

resident memory

The memory that is directly accessible by the CPU; the system's primary memory resource.

run queue

In OSF/1, a queue that contains threads that are ready to execute.

run-time registration

The configuration of devices into the kernel while the kernel is still up and running. *See also* **dynamic configuration**.

scheduler

1. The kernel subsystem that is responsible for scheduling threads for execution.
2. The layer of the LVM that schedules physical requests for logical operations and handles mirrors. *See also* **Logical Volume Manager (LVM)**.

scheduling priority

A per-thread attribute that is used to determine when the thread will next be scheduled for execution. In timesharing mode, the scheduler subsystem frequently adjusts each thread's priority so that all threads have approximately equal access to the system's CPU resources.

sector

1. An area on a disk track or a diskette track that is reserved to record information.
2. The smallest amount of information that can be written to or read from a disk or diskette during a single read or write operation.
3. On disk or diskette storage, an addressable subdivision of a track that is used to record one block of a program or data.
4. The smallest unit of I/O to a physical disk, and hence to a physical volume. This is typically 512 bytes. Sector is often a synonym for block. *See also* **physical volume**, **block**.

security classes

A means of classifying levels of security. The National Computer Security Center (NCSC) defines a set of security classes, ranging from A to D (with gradations within each class). The criteria for the general classes are as follows:

A Verified protection

B Mandatory protection

C Discretionary protection

D Minimal security

Within each class are subclasses that are indicated by a number, with higher numbers indicating higher security. That is, C2 offers more security than C1, and B1 offers more than C2. The classification is hierarchical, meaning that each class includes all of the features of the previous classes. The

significant levels for OSF/1 configurations are C2 and B1. The C2 level contains the following features:

- Individual password controls and auditing of security-related events.
- Access controls "capable of including or excluding to the granularity of a single user."[5] OSF/1 fulfills this requirement by using ACLs.
- Object reuse protection, ensuring that data left in memory, on disk, or elsewhere is not accessible to inappropriate users.

The B1 level contains the following features:

- Mandatory access control.
- Rigorous separation of security-related portions of the system from those portions that are not related to security.
- Additional testing and documentation, including a model of the security policy that is supported.

The next level, B2, requires additional assurance that the system cannot be penetrated. Some of the features that are required at B2 are already in place in OSF/1. For example, OSF/1 supports the requirement of *least privilege*, which stipulates that users and programs possess the least number of privileges possible, and for the least time necessary to perform operations. *See also* **Orange Book**.

security policy

Defined by the "Orange Book" as "A set of rules that are used by the system to determine whether a given subject can be permitted to gain access to a specific object."[6] The two security policies that can be configured into OSF/1 are

5. *Trusted Computer System Evaluation Criteria (TCSEC)* (CSC-STD-001-83), U.S. Department of Defense, National Computer Security Center, August 15, 1983. Section 2.2.1.1.
6. *Trusted Computer System Evaluation Criteria (TCSEC)* (CSC-STD-001-83), U.S. Department of Defense, National Computer Security Center, August 15, 1983. Requirement 1. Preface.

discretionary access control and mandatory access control. *See also* **discretionary access control (DAC), mandatory access control (MAC).**

send rights

A port right that allows a task to send messages on the port.

sensitivity label

A "combination of hierarchical classification levels and non-hierarchical categories"[7] (or compartments) that are assigned to subjects and objects, and are used as the basis for mandatory access control decisions. *See also* **mandatory access control (MAC).**

sensitivity level

Under mandatory access control, a classification of all information according to how sensitive that information is. This classification is called a sensitivity level. *See also* **sensitivity label, mandatory access control (MAC).**

Serial Line Internet Protocol (SLIP)

A transmission line protocol that encapsulates and transfers IP datagrams over asynchronous serial lines.

serial program

A program that performs its operations using a single thread of control. *See also* **parallel program.**

server

1. RPC: The party that receives remote procedure calls. A given application can act as both an RPC server and an RPC client. *See also* **client.**

7. *Trusted Computer System Evaluation Criteria (TCSEC)* (CSC-STD-001-83), U.S. Department of Defense, National Computer Security Center, August 15, 1983. Section 3.1.1.4.

2. CDS: A node that is running the CDS server software. A CDS server handles name-lookup requests and maintains the contents of the clearinghouse or clearinghouses at its node.
3. DTS: A system or process that synchronizes with its peers and provides its clock value to clerks and their client applications.
4. DFS: A provider of resources or services. *See also* **client**.
5. GDS: The server consists of a DSA, which accesses the database, and an S-stub, which handles the connection over the communications network for responding to remote clients and accessing remote servers.
6. An application program that usually runs in the background (daemon) and is controlled by the system program controller.
7. On a network, the computer that contains the data or provides the facilities that are to be accessed by other computers on the network.
8. A program that handles protocol, queuing, routing, and other tasks that are necessary for data transfer between devices in a computer system.
9. The component of the X Window System that manages input and the visual display.
10. The portion of a distributed program that handles requests for service from one or more client programs. The server's address space is separate from the client address spaces. *See also* **client**.

shadow chain

A chain of shadow objects that is the result of multiple symmetric copy-on-write operations. *See also* **shadow object**.

shadow object

A virtual memory object that is created when a task first attempts to write data that is shared with another task copy-on-write. Before the kernel allows the task to write the page,

it allocates a new page and physical resources, and creates a shadow object to manage the new page. If the task writes other pages in the original VM object, they are copied and inserted into the shadow object.

share map

A data structure that allows two or more tasks to share read/write access to data. The tasks must be related to one another.

shared library

A library that contains at least one subroutine that can be used by multiple processes. Programs and subroutines are linked as before, but the code that is common to different subroutines is combined in one library file that can be loaded at run time and shared by many programs.

Shift-Japanese Industrial Standard (SJIS)

An encoding scheme consisting of single bytes and double bytes that are used for character encoding. Because of the large number of characters in the Japanese and other Asian languages, the 8-bit byte is not sufficient for character encoding.

signal

1. Threads: To wake only one thread that is waiting on a condition variable.
2. A simple method of communications between two processes. One process can inform the other process when an event occurs.
3. In operating system operations, a method of inter-process communication that simulates software interrupts.
4. An interrupt that is generated by software that interrupts a process. *See also* **catch a signal**, **signal handler**.

signal handler

1. A process-specific routine that is invoked when the process receives a particular signal.
2. A subroutine that is called when a signal occurs.

socket

1. A port identifier.
2. A 16-bit port number.
3. A unique host identifier that is created by the concatenation of a port identifier with an IP address.
4. In interprocess communication, an endpoint of communication.

spl synchronization

A software exclusion method that is used to mask interrupts from hardware. On uniprocessor network code, this is often sufficient to implement protection of critical sections.

statically bound

The state of being semipermanently bound to the kernel. A statically bound device can only be configured into and unconfigured from the kernel when the kernel is not running.

stream head

The stream component that is closest to the user process, providing the interface to the user process.

STREAMS

1. A kernel mechanism from AT&T that supports the implementation of device drivers and networking protocol stacks.
2. A kernel mechanism from AT&T that supports the implementation of device drivers and networking protocol stacks.

STREAMS module

A set of routines that may be pushed into a stream to process control of data. It has a read queue and a write queue, and communicates with other elements of the stream by means of messages.

subject

Defined by the "Orange Book" as something (for example, a process) that "causes information to flow among objects or changes the system state."[8] *See also* **security policy**.

submap

An address map that manages a subrange of a larger address map. For example, the kernel map includes several submaps that are used by various kernel subsystems. A subsystem can lock its submap without having to lock the entire kernel map.

swapping

1. Temporarily removing an active job from main storage, saving it on disk, and processing another job in the area of main storage that was formerly occupied by the first job.
2. A process that interchanges the contents of an area of real storage with the contents of an area in auxiliary storage.
3. In a system with virtual storage, a paging technique that writes the active pages of a job to auxiliary storage and reads pages of another job from auxiliary storage into real storage.
4. In traditional UNIX, a memory management mechanism that forces the entire contents of a process's address space to secondary storage. In OSF/1, the swapping mechanism makes pages available for pageout but does not force them out of resident memory.

8. *Trusted Computer System Evaluation Criteria (TCSEC)* (CSC-STD-001-83), U.S. Department of Defense, National Computer Security Center, August 15, 1983. Glossary.

symmetric copy-on-write

A copy-on-write mechanism that allows tasks to share a copy of temporary data. When a task attempts to write data that is shared in this manner, the kernel allocates new physical memory, copies the data to the new memory, and places the corresponding virtual page in a shadow object. *See also* **temporary data**, **virtual page**.

synchronization

1. DTS: The process by which a DTS entity requests clock values from other systems, computes a new time from the values, and adjusts its system clock to the new time.
2. The fundamental mechanism of locking between different threads of execution in the kernel and between the kernel and the interrupt handlers.

system call

1. A request by an active process for a service by the system kernel.
2. A request by an active process for a service by the system kernel. A process executing a system call generates an exception that traps the process into the kernel so that it may run in kernel mode.

tag

Kernel encodings of policy-specific security information. A tag pool is associated with each UNIX data structure that describes a subject or object. The tag pool contains the tags that represent the security attributes for that subject or object. Note that tags and privilege sets in the file system are independent. Privileges reside in privilege vectors, not in tags.

Tags are maintained in a security policy database that is managed by policy daemons. Each policy module maintains a decision cache. If the requisite information is not in the cache, the system must get its decision (and possibly a new tag) from a policy daemon.

task

1. A basic unit of work that is to be performed. Some examples include a user task, a server task, and a processor task.
2. A process and the procedures that run the process.
3. In a multiprogramming or multiprocessing environment, one or more sequences of instructions that are treated by a control program as an element of work to be accomplished by a computer.
4. A data structure that represents a set of system resources that provides a context for the execution of one or more threads. These resources include a virtual address space and Mach IPC ports. *See also* **thread**.

task exception

A program exception that is task-specific, not thread-specific. *See also* **program exception**, **thread exception**.

temporary data

Data that is generated in resident memory as a process executes. For example, the data contained in a process's heap is temporary; when the process exits, the heap data is lost. *See also* **permanent data**.

terminal

1. A device, which is usually equipped with a keyboard and a display device, that is capable of sending and receiving information over a communications line. *See also* **tty**.
2. In a system or communications network, a point at which data can either enter or leave.
3. In curses, a special screen that represents what the work station's display screen currently looks like.

text

1. A type of data consisting of a set of linguistic characters (letters, numbers, and symbols) and formatting controls.

2. The executable portion of a program, as contained within an object file, or as loaded into memory. Operating systems generally make the text read-only, and generally arrange for multiple processes to share a single text image. *See also* **object file format**, **bss**, **data section**.
3. In kernel mode, contains kernel program code that executes. It is read-only by a user process.
4. In ASCII and data communications, a sequence of characters that is treated as an entity when preceded by one start-of-text and terminated by one end-of-text communication control character.
5. In word processing, information that is intended for human viewing and that is presented in a two-dimensional form, such as data printed on paper or displayed on a screen.
6. The part of a message that is not the header or control information.

thrashing

In a virtual storage system, a condition in which the system is doing so much paging that little useful work can be done.

thread

1. A single, sequential flow of control within a process.
2. A single, sequential flow of control.
3. An independent computation that operates within the same context as other independent computations.
4. A data structure that represents an independent computation.

thread exception

A program exception that is thread-specific, not task-specific. For example, a divide-by-zero exception is a thread exception. *See also* **program exception**, **task exception**.

TLB

See **translation lookaside buffer (TLB).**

TLB shootdown

An operation that invalidates the contents of a CPU's translation lookaside buffer. *See also* **translation lookaside buffer (TLB).**

translation lookaside buffer (TLB)

A cache on the CPU that contains recently used address translations. When performing address translation, the memory management unit searches the TLB before searching page tables in memory. The TLB significantly optimized address translation operations. *See also* **memory management unit (MMU).**

trusted computing base (TCB)

Defined by the "Orange Book" as "The totality of protection mechanisms within a computer system—including hardware, firmware, and software—the combination of which is responsible for enforcing a security policy. A TCB consists of one or more components that together enforce a unified security policy over a product or system. The ability of a TCB to correctly enforce a security policy depends solely on the mechanisms within the TCB and on the correct input by system administrative personnel of parameters (for example, a user's clearance) related to the security policy."[9] *See also* **Orange Book**.

tty

Any device that uses the **termios** standard terminal device interface. The tty devices typically perform input and output on a character-by-character basis.

9. *Trusted Computer System Evaluation Criteria (TCSEC)* (CSC-STD-001-83), U.S. Department of Defense, National Computer Security Center, August 15, 1983. p. 116.

UID

See **user ID**.

uniprocessor

A hardware platform that contains one CPU.

user data area

The system, virtual memory paging space, or application data. *See also* **Logical Volume Manager (LVM)**, **physical volume**.

user ID

An integer that uniquely identifies a system user.

user mode

1. A mode in which a process is carried out in the user's program rather than in the kernel. Contrast with **kernel mode**.
2. A mode of execution in which the CPU executes user instructions, but does not have access to kernel instructions and data. *See also* **kernel mode**.

user stack

A region in a process's virtual address space that contains local variables that are being used by a currently active subroutine. When a subroutine calls another subroutine, the kernel allocates a new stack frame to hold the new routine's variables and pushes the stack frame on the user stack.

validity fault

A page fault that is generated when a thread attempts to reference a page that is not in resident memory. *See also* **page fault**, **page fault handler**, **protection fault**.

VFS

See **virtual file system (VFS)**.

virtual address space

A contiguous range of virtual memory. In UNIX systems, each process has a single virtual address space that contains the process's executable text and data. *See also* **address translation**.

virtual file system (VFS)

1. DFS: A level of abstraction that is above the specific interfaces to various types of file systems. It is used to avoid having to change kernel code to handle low-level, system-specific differences.
2. In OSF/1, a kernel subsystem that implements a level of abstraction that is above the specific interfaces to various types of file systems.

virtual memory

Addressable space that appears to be real memory. From virtual memory, virtual addresses are mapped into real memory locations. The size of virtual memory is limited by the addressing scheme of the computer system and by the amount of auxiliary storage that is available, not by the actual number of main storage locations.

virtual memory object (VM object)

A kernel data structure that represents a set of virtual pages that are mapped into one or more virtual address spaces. *See also* **memory object**.

virtual page

A software abstraction that the kernel uses to manage the system's memory resources. Each system has a system-specific virtual page size that either matches the hardware's page frame size or is a multiple of two of that size. *See also* **page frame**.

VM object

See **virtual memory object (VM object)**.

vnode

1. DFS: The structure that is used to access the inode or anode structure that is associated with a specific file through a virtual file system interface. The term vnode stands for virtual node.
2. A data structure that is used by the kernel to manage a file or directory. There is a unique vnode that is allocated for each of the system's active files and directories. Each vnode represents an underlying, file-system-specific data structure. The vnode construct allows the kernel's virtual file system to manage different file systems through a uniform interface. *See also* **memory object**.

vnode pager

The kernel's default pager, and the pager that manages paging operations on mapped files. The vnode pager is so named because the memory objects it manages are represented by vnodes. *See also* **memory object**, **vnode**.

volume

1. A certain portion of data, together with its data carrier, that can be handled conveniently as a unit.
2. The level of sound of the system.
3. The physical storage location of a file system.
4. A block storage device that corresponds to a disk driver, or a disk partition in a traditional UNIX system. *See also* **Logical Volume Manager (LVM)**.

volume group

1. A collection of physical volumes (read/write hard drives) of varying sizes and types.
2. A set of physical and logical volumes, and the mappings between them. Logical volumes can only map physical volumes that are in the same volume group. *See also* **physical volume**, **logical volume**, **Logical Volume Manager (LVM)**.

wide character

The C type definition **wchar_t** that is used to store process codes in a program.

Index

Symbols

__**lc_load**( ) interface, 10–10

A

a.out, 8–5
ablsolute executable image, 8–1
absolute
- load module, 8–3, 8–4
- object file region, 8–18

access control list (ACL), 15–2 to 15–6, 15–24, 15–26 to 15–30
ACL. *See* access control list
active page queue, 7–8
adding protocols dynamically, 12–5
address map
- and entries, 6–2, 6–8
- and memory objects, 3–8
- and pmap, 3–16
- and regions, 3–9
- introduction, 3–3
- kernel, 6–29, 6–31
- locks, 6–29
- task, 3–10
- task swapping, 7–16

address map entry, 6–7, 6–8, 6–30
address space
- and copy map object, 6–29
- and memory objects, 3–8, 3–12
- and pmap, 6–34
- and the program loader, 8–1 to 8–11, 8–18, 8–20
- fast lookup, 6–2
- implementation, 3–10, 6–1, 6–8, 6–9
- in traditional UNIX, 3–16, 3–19
- kernel, 6–29
- locks, 6–30
- primitives that manipulate, 3–17, 3–18
- process, 2–2
- regions, 6–3
- sparsely filled, 3–9

address translation, 6–32, 6–33, 6–37
allocation of file descriptors, 11–7
allocb( ) routine, 13–19
alphabetic sorting, 10–1
API, 10–9, 14–12

application programming interface.
    See API
application programs,
    internationalization, 10–1 to
    10–12
Asian code sets, 10–14
assert_wait( ) routine, 5–14, 5–16
asymmetric copy-on-write, 6–10,
    6–25, 6–26, 6–29
asynchronous signal, 4–7, 4–11,
    4–16
audit, 15–10, 15–44 to 15–56
    and locks, 15–40
    and LUID, 15–38
    compaction files, 15–50
    control files, 15–36
    daemon, 15–47
    data structures, 15–48
    device driver, 15–46
    internal kernel buffering,
        15–49
    ISSO interface, 15–47
    pathname processing, 15–55
    record format, 15–51
    record generation, 15–53
    reduction program, 15–47
audit_info structure, 15–68
authentication, 15–7, 15–20,
    15–36, 15–37 to 15–44
authorizations
    command, 15–3, 15–19,
        15–20, 15–43
    kernel, 15–3, 15–20, 15–21
automatic configuration, 9–3

B

B1 certification, 15–2, 15–4
backing object, 6–13, 6–16, 6–17
bad blocks, 14–2, 14–8, 14–21,
    14–26, 14–27
bad sectors, 14–2, 14–8, 14–26,
    14–27
base privilege set, 15–3, 15–20,
    15–65
binding (or linking), 8–1, 8–3, 8–9
block device, 11–12, 11–25
block-oriented storage, 14–2
boot time configuration, 9–10
bottom-half configuration, 9–6
BSD scheduler, 5–2 to 5–4
bufcall( ) routine, 13–19, 13–20
buffer
    cache, 11–12, 11–19, 11–27
    free list, 11–28

C

C2 certification, 15–2, 15–4
cache
    buffer, 11–12, 11–19, 11–27
    name, 11–16
    preload, 8–22
cache management algorithm,
    11–16
callback, 13–9, 13–11, 13–19
    requests, 13–1
canput( ) routine, 13–16
catgets( ) function, 10–11

catopen( ) function, 10–11
cdevsw table, 13–7
cdevsw_open_comm( ) routine, 13–22
certification, 15–2, 15–4
character
  classification, 10–6
  device revocation, 11–24
  devices, 11–25
  longer than one byte, 10–3
classification of characters, 10–6
clean buffer list, 11–19
cleaning-in-place and cluster pageout, 7–11
clearance, 15–3, 15–6 to 15–8, 15–30 to 15–34, 15–36
client parallelization, 11–31
clock interrupt handler, 2–10, 5–3, 5–4
clone device, 13–7, 13–21
clone devices, 11–27
cloning, 13–21
cluster data structure, 12–21, 12–22
cluster size, 7–5
coarse-grained application scheduling, 5–23
code sets
  Asian, 10–14
  converting between, 10–15
  eight-bit, 10–3, 10–12
  multibyte, 10–3
  single-byte, 10–3
COFF, 8–5
collation, 10–4
command
  authorizations, 15–3, 15–19, 15–20
  authorizations database, 15–39, 15–43
commands, **autopush**, 10–18
Common Object File Format (COFF), 8–5
compare-and-swap locks, 5–23
computation-bound program, 5–3
computational state, 2–8, 3–2
configuration
  automatic, 9–3
  boot time, 9–10
  bottom half, 9–6
  database, 9–3
  dynamic, 14–2, 14–18
  manager, 9–3
  method, 9–3
  of dynamic system calls, 9–9
  of file system types, 9–8
  security, 15–6
  selective, 9–10
  static, 9–10
  top half, 9–6
context switching
  and discouragement hints, 5–23
  and quantum, 5–2
  and the **thread_block**( ) routine, 5–11, 5–14
  and the **thread_switch**( ) routine, 5–24
  and timers, 5–27
  introduction, 2–8
control device, 14–19, 14–24

conventions for date and time, 10–6
conversion table, 10–16
converting between code sets, 10–15
copied region, 6–9
copy map object, 6–29
copy object, 6–22, 6–24, 6–26, 6–29
copy-on-write, 3–6, 3–9, 6–10
  asymmetric, 6–10, 6–25, 6–26, 6–29
  symmetric, 6–10, 6–14, 6–22, 6–26
covered vnode, 11–11

CPU-usage timers, 5–2, 5–25, 5–26
csq_acquire( ) routine, 13–13, 13–25
csq_lateral( ) routine, 13–13, 13–17

D

data
  mirroring, 14–6
  permanent, 6–22, 6–23, 6–26, 7–3
  recovery, 14–6
  replication, 14–2, 14–6
  restoring, 14–11
  stale, 14–22
  temporary, 6–10, 6–14, 6–22, 7–4
date and time conventions, 10–6
date display, international, 10–1
deadlocks, 12–14, 13–16
deallocation of file descriptors, 11–7
default
  pager, 3–13, 7–3
  pager, submap, 6–30
  processor set, 5–19, 5–20, 5–21
deferred catalog opens, 10–11
deleting protocols dynamically, 12–5
demand paging, 2–6, 7–1
descriptor management, 11–3
device
  assignment database, 15–36, 15–39, 15–43, 15–44
  driver, 2–11
  driver, bottom half, 9–6, 9–7
  driver, dynamically configurable, 9–6
  driver, top half, 9–6, 9–7
  hashing, 11–26
  interrupt, 2–12
  special files, 11–25
devices, logical volumes, 14–4
dirty buffer list, 11–19
discretionary access control (DAC), 15–7, 15–11, 15–24 to 15–30, 15–68
disk drivers, 14–2
disks, 14–1, 14–2, 14–4, 14–8
distributed file system, 11–15
domain, 12–3
  families, 12–2
  funnel mechanism, 12–10, 12–13, 12–18

list, 12–5
reference count, 12–5
domain data structure, 12–3, 12–4
DOMAIN_FUNNEL macro, 12–19, 12–21
domain_funnel structure, 12–19, 12–21
DOMAIN_UNFUNNEL macro, 12–19, 12–21
dynamic
configuration, 11–38, 14–2, 14–18
configuration changes, 11–32
device driver, 9–6
loading, 8–2, 8–12, 8–13, 8–18, 8–23
system call numbers, 9–9
system calls, 9–9
unconfiguration, 14–18

E

effective privilege set, 15–3, 15–21, 15–22, 15–32, 15–64, 15–65
eight-bit code sets, 10–3
esballoc( ) routine, 13–20
event management, 12–9
event-wait mechanism, 5–13, 5–14, 5–16, 5–17
events, 13–8
exception handler, 4–13, 4–15
default, 4–14
exception handling facility, 4–13 to 4–16
exec, 4–6
exec switch, 8–6, 8–7
exec( ) system call
algorithm, 8–7
and absolute images, 8–1
and **ld** in UNIX, 8–5
and privileged processes, 8–25
and UNIX processes, 4–1
architecture, 8–6
introduction, 2–3
executable image, absolute, 8–1
execution
mode, 2–10
state, 2–4, 3–2, 5–8 to 5–18
exit( ) system call, 2–3, 4–1, 4–6
exported
packages list, 8–14
symbol, 8–5, 8–9, 8–10, 8–12, 8–17
extent size, 14–4
external
data, 12–25
memory manager, 3–13, 7–3, 7–17, 7–19
memory manager and paging, 7–9
memory manager interface, 7–3, 7–19
pager, 6–27
reference, 8–1, 8–2, 8–3, 8–4, 8–10
VM objects, 3–12
extra privilege checking, 11–39

F

fast symbolic links, 11–35
fattach( ) routine, 13–21
fictitious pages, 7–10
file
    format manager, 8–6, 8–7, 8–8, 8–17, 8–18, 8–23
    layer, 11–3
    locking, 11–24
file descriptor, 11–7, 12–1
    state, 2–4
    table, 4–2, 4–4
file system
    layer, 11–30
    metadata, 11–25
    private data, 11–20
    security extensions, 11–38, 15–56 to 15–61
    tree, 11–9
    type, loading, 9–8
fine-grained application scheduling, 5–23
flow control, 13–1
fmodsw table, 13–7
forcible unmount, 11–11, 11–24
fork( ) system call, 2–3, 4–1, 4–6, 13–21
format-dependent loader routines, 8–17
framework, 13–1
free list chains, 11–28
free page queue, 7–8
free( ) routine, 2–2
freeb( ) routine, 13–20
funneling, 11–36

G

general registers, 2–9
getaddressconf( ) system call, 8–20
getmsg( ) routine, 13–5
getnewvnode( ) routine, 11–23
getpmsg( ) routine, 13–5
global
    data file, 8–20, 8–22
    installed packages table (global IPT), 8–11, 8–20, 8–22, 8–23, 8–25
granted privilege set, 11–39, 15–3, 15–21, 15–22, 15–60
group ID, 2–5

H

hardware clock, 2–12

I

iconv conversion subsystem, 10–15
iconv( ) converters, 10–10
iconv( ) function, 10–15
iconv_close( ) function, 10–15
iconv_open( ) function, 10–15
identification, 2–5, 15–7, 15–42

imported
  packages list, 8–14
  symbol, 8–5, 8–8, 8–9, 8–10, 8–12, 8–14, 8–15, 8–24
inactive page queue, 7–8
information systems security officer (ISSO), 15–19, 15–20, 15–23, 15–44, 15–47
inlib build-in shell command, 8–11
inode locking, 11–37
installing libraries, 8–11, 8–22
interactive program and CPU, 5–3
internal VM objects, 3–12
internationalization subsystem, 10–1 to 10–12
Internet
  domain, 12–2
  domain locks, 12–16
  protocols, 12–17
interrupt
  handler, 2–12
  handler, clock, 2–10, 5–3, 5–4
  handler, registering, 9–6
  level, 2–12
  service routine, 12–5
  service threads, 12–8, 12–13
interrupts, 13–16
invalidation of vnodes, 11–24
ioctl( ) routine, 13–5
IPC and paging requests, 7–2, 7–3, 7–10, 7–11
ISR threads, 12–8, 12–13

J

job control facility, 4–1, 4–7

K

keep-on-exec, 8–25
kernel
  authorizations, 15–3, 15–20, 15–21
  daemons, 2–13
  load server, 8–21
  map, 6–29
  mode, 2–10
  space loading, 8–21
  task, 6–29
  timer, 5–26, 5–27
  UNIX, 2–1, 2–3, 2–10
kmodcall( ) system call, 9–3
known module, 8–9
known modules list, 8–14, 8–15, 8–16

L

Latin-1, 10–13
lazy evaluation, 3–6, 3–17, 6–32, 6–36, 7–6
ld command, 8–3, 8–4, 8–5, 8–10
least privilege principle, 15–4

least recently used (LRU)
name cache policy, 11–16
paging policy, 7–9
libraries, installing, 8–11, 8–22
lib_admin command, 8–11, 8–19, 8–22
linker, 8–3, 8–5, 8–10
load average calculation, 5–3, 5–5
load( )
interface, 8–12
system call, 8–24
load-time linking, 8–4
loaded packages table (LPT), 8–12
loader
address space management, 8–18
context, 8–8, 8–12, 8–13, 8–18
format-dependent routines, 8–17
memory allocation interfaces, 8–19
security, 15–65
switch, 8–8, 8–14, 8–17
loading
a file system type, 9–8
kernel space, 8–21
locale, 10–1
locking, 12–14, 13–9
locks
compare-and-swap, 5–23
test-and-set, 5–23
user address map, 6–29
logical
address space, 2–2
addresses, 2–7
block requests, 14–21, 14–23
extents, 14–4, 14–6
page, 3–14
track group, 14–5, 14–21
Logical Volume Manager. *See* LVM
logical volumes, 14–2, 14–4
allocation unit, 14–4
and physical volumes, 14–2, 14–6, 14–8
defect mirroring, 14–27
logical extents, 14–4
LTG. *See* logical track group
LVM, 14–1 to 14–27

M

Mach
Interprocess Communication (Mach IPC), 3–4, 4–13, 6–29
security, 15–66
machine-dependent, virtual memory management, 3–16, 3–17
machine-independent, 6–36, 6–37
data structures, 6–32
resident memory management, 3–14
virtual memory management, 3–16, 3–17
malloc( ) routine, 2–2, 12–25
managing disk storage, 14–1
mandatory access control (MAC), 15–2 to 15–7, 15–11, 15–30 to 15–37, 15–62, 15–63, 15–67

mapping, logical-to-physical, 14–6
masking signals, 4–10
mbuf
chain, 12–15, 12–22
data structure, 12–21, 12–22, 12–23, 13–20
MCLALLOC macro, 12–25
MCLGET macro, 12–25
memory
allocation, 13–1, 13–19
allocation interfaces, loader, 8–19
external manager, 3–13, 7–3, 7–17, 7–19
external manager and paging, 7–9
management hybrid policy, 7–2
management unit (MMU), 2–7, 6–32, 6–37
manager, 7–2
map, 4–2, 4–3
mapping, 2–6
object, 3–1, 3–6, 3–8 to 3–13, 3–16, 3–18, 3–19
object, cache management interface, 7–20
physical, 3–14
region, 3–9, 3–11, 3–18, 3–19, 6–2 to 6–9, 6–13, 6–27
region, and memory object, 7–2
resident, 2–6, 7–1
memory_object_data_provided( ) routine, 7–20
message, 3–6, 3–7
and paging, 3–12
definition, 3–4
subsystem, 10–11
used to invoke an operation, 3–8
MFREE macro, 12–25
MGET macro, 12–25
MGETHDR macro, 12–25
minor number space, 13–22
mirror consistency record, 14–9, 14–10, 14–25
mirroring, 14–6
consistency, 14–22
data in transition, 14–23
doubly, 14–6
logical volume defects, 14–27
physical extents, 14–6
recovery, 14–6
replication, 14–22
restoring, 14–11
resynchronization, 14–22
sequential policy, 14–24
singly, 14–6
stale data, 14–22
synchronization, 14–21
mknod( ) routine, 13–21
mmap( ) routine, 6–28, 7–3
and loader, 8–20
MMU, 2–7, 6–32, 6–37
module record, 8–14, 8–15, 8–23
monetary display, international, 10–1
monetary formatting, 10–8
mount point, 11–13
mount structure, 11–11
mounted
block device, 11–12
file system, 11–10, 11–14

mpsleep( ) routine, 13–18
multibyte
code sets, 10–3
encoding methods, 10–3
multilevel directories, 11–40
multiplexing streams, 13–23, 13–24
multiprocessor environment, 12–13
multiprocessor networking code, 12–13

N

name
cache, 11–16
translation, 11–12
namei( ) function, 11–13
National Computer Security Center (NCSC), 15–1, 15–4
negative caching, 11–16
Netintr( ) routine, 12–9, 12–13
netisr data structure, 12–10, 12–12
netisr framework, 13–5, 13–8
netisr_add( ) routine, 12–9
netisr_del( ) routine, 12–9
netisr_input( ) routine, 12–12
NETISR_STREAMS event, 13–8
NETISR_STRTO event, 13–8
NETISR_STRWELD event, 13–8
netisr_thread event, 12–13
nonresident page, 3–11
nonparallelized protocol, 12–2, 12–13, 12–18
np_uthread structure, 4–2, 4–4, 4–5
numeric display, international, 10–1

O

object
backing, 6–13, 6–16, 6–17
copy, 6–22, 6–24, 6–26, 6–29
copy map, 6–29

object file, 8–2, 8–3, 8–4, 8–5, 8–15, 8–18
format, 8–2, 8–4, 8–5, 8–6
object-oriented subsystem, internationalization, 10–9
operations vector, 11–11
operator, 15–22, 15–23
Orange Book, 15–1, 15–4, 15–7, 15–10
OSF/ROSE, 8–5
OSR, 13–7
OSRQ, 13–7
out-of-line data, 3–6

P

packages, 8–9, 8–10, 8–11, 8–14, 8–17
packet processing, 12–12
page
caching, 3–15, 3–20, 6–35, 7–17, 7–20
cluster size, 7–5
clustering, 7–5 to 7–8
fault, 3–11, 6–3, 6–8, 6–26, 6–30, 6–34, 6–37
fault exception, 2–7
fault handler, 2–7, 6–7, 6–26, 7–2, 7–3, 7–9

fault handler and page
clustering, 7–5, 7–12
fictitious, 7–10
frames, 2–6, 3–14
logical, 3–14
nonresident, 3–11
private, 7–11
queue, 7–8
replacement algorithm, 7–8
resident, 3–11, 3–14, 3–15
size, 14–5
target, 7–8, 7–11, 7–12
wired, 3–15
pagein
and caching, 3–20
and memory objects, 3–9
of clusters, 7–7, 7–12
requests, 7–2
thread protection, 7–13
vnode pager, 7–3
pageout, 3–9, 3–13, 3–20
pageout daemon, 2–8, 2–13, 7–5,
7–6 to 7–9, 7–10, 7–11,
7–14
and scheduler, 5–4
pager
default, 6–30, 7–3
external, 6–27
trusted, 7–3
pager port, 3–11
pager_file data structure, 7–6
paging
demand, 2–6, 7–1
files, 7–4 to 7–7
manager, 3–8
parallel programming models and
the scheduler, 5–1, 5–18
parallel protocol, 12–14
parallelization, 11–30
of NFS server, 11–31
parallelized protocol, 12–2
patching (subroutine and global
variable references), 8–3,
8–5
pathname translation, 11–12
pcb structure, 12–14, 12–16
per-process open-file table, 11–3
permanent data, 6–22, 6–23, 6–26,
7–3
pf_drain( ) routine, 12–25
physical
disks, spanning, 14–2
extents, 14–4, 14–6
file system, 11–12
map (pmap), 3–16 to 3–17,
6–32 to 6–39, 7–9,
7–10, 7–15, 7–16
memory, 3–14
volumes, 14–2 to 14–10,
14–19, 14–26
physical volume reserved area,
14–8
PID, 2–5
pipes, 13–21
pmap, 3–16 to 3–17, 6–32 to 6–39,
7–9, 7–10, 7–15, 7–16
functions, 6–32 to 6–37
port
and memory object, 3–9
as object, 3–8
definition, 3–4
name space, 3–5
rights, 3–5 to 3–6
potential privilege set, 11–39,
15–3, 15–21, 15–60

predetermined system call
numbers, 9–9
preload, 8–13, 8–17, 8–19, 8–22
preload cache, 8–22
prerelocate, 8–22
private installed packages table
(private IPT), 8–11, 8–25
private page, 7–11
privilege bracketing, 15–4
privileges, 15–3, 15–18 to 15–23,
15–28, 15–32
probing devices, 9–10
proc structure, 2–5, 4–1, 4–3
proc structure (UNIX), 4–5, 5–3
process
context, 2–8
ID (PID), 2–5
relation state, 2–5
UNIX, 2–1
processor
as object, 5–19, 5–20
data structure, 5–19
server program, 5–19, 5–21
processor set, 5–18, 5–19 to 5–23
as object, 5–19, 5–20
data structure, 5–19
default, 5–19, 5–20, 5–21
program counter, 2–8
program exceptions, 2–11
protected password database,
15–36, 15–39 to 15–41,
15–43
protocol, 12–3
control block, 12–7, 12–16
deleting dynamically, 12–5
endpoints, 13–21
Internet, 12–17
nonparallelized, 12–2,
12–13, 12–18
parallelized, 12–2, 12–14
protocol-to-protocol service
requests, 12–3
protosw data structure, 12–3
pseudodevices, 9–7
pseudoterminals, 13–21
pse_select( ) routine, 13–5
psignal( ) (UNIX), 4–12
psignal_indirect( ) routine, 4–12
psignal_internal( ) routine, 4–12
putmsg( ) routine, 13–5
putnext( ) routine, 13–13
putpmsg( ) routine, 13–5
PVRA. *See* physical volume
reserved area

Q

qenable( ) routine, 13–16, 13–19,
13–20, 13–23
qreply( ) routine, 13–13
quantum (time-slice), 2–13, 5–2,
5–3
queue
active page, 7–8
free page, 7–8
inactive page, 7–8
wait, 5–13, 5–14, 5–15,
5–16
queue-pair synchronization, 13–28
queued messages, 13–17
quorums, 14–6, 14–9, 14–10,
14–19, 14–25

q_next routine, 13–22
q_qlock routine, 13–16

R

raw disk paging file, 7–4
receive rights, 3–5
recursion, 13–11
recycling vnodes, 11–23
region
 copied, 6–9
 in object file, 8–4, 8–8, 8–17, 8–18, 8–21, 8–23, 8–25
 memory, 6–2 to 6–9, 6–13, 6–27
 memory, inheritance, 3–18
 memory, interfaces, 3–19
 memory, introduction, 3–9
 memory, tracking, 3–11
 record, 8–14, 8–23
 shared, 6–9
registering interrupt handlers, 9–6
registers, general, 2–9
relocatable object file region, 8–18
relocation
 by the loader, 8–3, 8–5, 8–8, 8–12, 8–14, 8–18, 8–22
 sectors, 14–9
 software, 14–9
remote procedure call. *See* RPC
request
 unweld, 13–8
 weld, 13–8, 13–16, 13–22
resident
 memory, 2–6, 7–1, 7–2, 7–8, 7–20
 page, 3–11, 3–14, 3–15
 page table, 3–15, 7–10, 7–13
resynchronization, 14–10
root, 15–2, 15–3, 15–11, 15–22, 15–41, 15–42, 15–65
RPC, 3–8, 4–13
run queue
 and context-switching, 5–2
 and event-wait, 5–13
 and load, 5–4
 and priority updates, 5–7
 and suspend state, 5–10, 5–12
 and thread state, 5–8
 data structure, 5–7

S

sbappend( ) routine, 12–15
schednetisr( ) macro, 12–9
schednetisr( ) routine, 12–12
scheduler
 BSD, 5–2 to 5–4
 OSF/1, 5–4 to 5–7
 timestamp, 5–2, 5–5, 5–7, 5–26, 5–28
 usage-aging mechanism, 5–3, 5–6
scheduling
 hints, 5–18, 5–23 to 5–25
 network activity, 12–8

priority, 5–2, 5–8, 5–25
state, 2–4
secondary storage, 7–1
sectors, relocating, LVM, 14–9
secure systems, 9–10
security, 15–1 to 15–69
conditionals, 15–6, 15–9, 15–18, 15–19, 15–21, 15–23
security policy, 15–1, 15–7 to 15–17
daemon, 15–11, 15–13, 15–15
database, 15–11, 15–14, 15–16
driver, 15–13, 15–16
module, 15–11, 15–13, 15–14, 15–24
switch, 15–11, 15–13, 15–24
selective configuration, 9–10
send rights, 3–5
sensitivity
label, 15–3, 15–4, 15–10, 15–30, 15–31
level, 15–3, 15–6, 15–7, 15–30, 15–34
server parallelization, 11–31
setlocale( ) function, 10–10
shadow
object, 6–12, 6–13, 6–15, 6–16, 6–22
object, chains, 6–15 to 6–18
trees, 6–19 to 6–21
share map, 6–27, 6–28
shared
file description access, 11–5
library, 8–1 to 8–5, 8–10, 8–11, 8–25
library, prerelocated, 8–22, 8–23
memory server, 7–17
region, 6–9
sharing sockets, 11–5
Shift Japanese Industrial Standard, 10–14
sigaction( ) system call, 4–10, 4–17
signal
and trap handler, 2–11
asynchronous, 4–7, 4–11, 4–16
delivery, 4–9
facility, 4–1, 4–7 to 4–12, 4–16
masking, 4–10
synchronous, 4–7, 4–13, 4–16
sigprocmask( ) system call, 4–10
sigsuspend( ) system call, 4–11
single-byte code sets, 10–3
single-queue synchronization, 13–28
singly mirrored, 14–6
sleep interruptibly, 5–13, 5–15
sleep( ) routine, 13–18, 13–19, 13–25
sockbuf structure, 12–7
socket, 11–3
framework, 12–2, 13–8
in kernel space, 12–1
in user space, 12–1
locks, 12–14
programming interface, 12–2
security extensions, 12–26, 15–63 to 15–65

socket data structure, 12–1, 12–6
socket( ) system call, 11–5
socket-to-protocol service requests, 12–3
SOCKET_LOCK macro, 12–15
socklocks structure, 12–15
soreceive( ) routine, 12–15
sorting, alphabetic, 10–1
sosbwait( ) routine, 12–15
sosleep( ) routine, 12–15
specalloc( ) routine, 11–25
special files, 11–24
spl synchronization, 12–14, 12–18
splnet( ) function, 12–21
splx( ) function, 12–21
stack management registers, 2–9
static configuration, 9–10
static linking, 9–10
sth_rput( ) routine, 13–25
storage, block-oriented, 14–2
stream
    head, 13–1, 13–3
    head routines, 13–7
    queue scheduling, 13–1
STREAMS, 13–1 to 13–29
    security extensions, 15–61 to 15–63
streams_mpsleep( ) routine, 13–18
streams_open_comm( ) routine, 13–22
strmod_add( ) routine, 13–27
strqset( ) routine, 13–17
submap
    of kernel address map, 6–30
    vnode pager as, 7–3
superblock changes, 11–40
superuser, 15–20, 15–23
suspend mechanism, 5–9, 5–10, 5–12
swapin daemon, 2–13
swapon command, 7–4
swapon() system call, 7–4
swapout daemon, 2–13
swapping, 2–6, 7–1, 7–13 to 7–16
    in OSF/1, 7–14
    tasks, 7–16
    threads, 7–15
symbol resolution, 8–9, 8–11, 8–14, 8–17
symbolic link, 11–13, 11–14
symmetric copy-on-write, 6–10, 6–14, 6–22, 6–26
synchronization, 12–10, 12–13, 13–9, 13–27
    on descriptors, 11–7
    queue, 13–9
    queue element, 13–11
    queue head, 13–11
    queue header, 13–28
synchronous signal, 4–7, 4–13, 4–16
syscalls.master file, 9–9
sysconfig command, 8–21
system administrator, 15–20, 15–23, 15–30, 15–34, 15–49
system call interface, 2–10
system call numbers
    dynamic, 9–9
    predetermined, 9–9
system calls
    dynamic configuration of, 9–9
    dynamic loading of, 9–9
    introduced, 2–10

system crash, 14–10, 14–22
system defaults database, 15–36, 15–39, 15–41, 15–43 to 15–44
System High, sensitivity label, 15–34
System Low, sensitivity label, 15–34
System V File System, 11–35
sysv_fs_configure( ) routine, 11–38

T

table
- conversion, 10–16
- file descriptor, 4–2, 4–4
- **fmodsw**, 13–7
- global installed packages, 8–11, 8–20, 8–22, 8–23, 8–25
- pre-process open-file, 11–3
- private installed packages, 8–11, 8–25
- resident page, 7–10, 7–13

tag, 15–14, 15–15, 15–16, 15–24, 15–29
tag pool, 15–14, 15–27, 15–28, 15–58, 15–60
tagged file systems, 11–38
target page, 7–8, 7–11, 7–12
target threshold, and swapper, 7–14
task
- address space, 3–9
- and thread creation, 3–6
- definition, 3–2
- exception port, 4–14, 4–15
- intertask communication, 3–4, 3–7
- port rights, 3–5
- swapper, 7–14, 7–15, 7–16

task data structure, 3–3, 4–2, 4–3, 4–5
TCB. *See* trusted computing base
TCP protocol, 12–15
TCP/IP domain, 12–2
tcp_input( ) routine, 12–15
temporary data, 6–10, 6–14, 6–22, 7–4
temporary memory object, 7–3, 7–7, 7–10
terminal control database, 15–39, 15–41 to 15–42, 15–43
test-and-set locks, 5–23
thrashing, 7–2
thread
- definition, 3–2
- exception clear port, 4–15
- exception facility, 4–13
- exception port, 4–15
- execution states, 5–8 to 5–18
- kernel stack, 5–9, 7–7, 7–13, 7–15
- swapper, 7–15

thread data structure, 3–3, 4–2, 4–5
thread_block( ) routine, 5–9, 5–11, 5–12, 5–14, 5–18, 5–24
thread_depress_priority( ) routine, 5–23, 5–24
thread_doexception( ) routine, 4–16

thread_hold( ) routine, 5–9, 5–11
thread_release( ) routine, 5–9, 5–11
thread_resume( ) routine, 5–9
thread_suspend( ) routine, 5–9
thread_switch( ) routine, 5–23, 5–24
thread_timeout( ) routine, 5–24
thread_wakeup( ) routine, 5–15, 5–16
time-sampling mechanism, 5–4, 5–25
time-slice, 2–13
timeout( ) routine, 13–19, 13–20
timer
  CPU-usage, 5–2, 5–25, 5–26
  data structure, 5–27
  kernel, 5–26, 5–27
  user, 5–26, 5–27
timesharing, 5–2 to 5–8
timestamp-based timing facility, 5–2, 5–5, 5–7, 5–26, 5–28
TLB, 6–37 to 6–39
top-half configuration, 9–6
translation lookaside buffer (TLB), 6–37 to 6–39
trap handler, 2–11
trusted computing base (TCB), 15–5, 15–7 to 15–11, 15–34, 15–45
trusted pager, 7–3
tsleep( ) routine, 13–18

U

u-area, 4–17
UFS implementation, 11–32
unconfiguration, dynamic, 14–18
union of pointers, 11–20
uniprocessor environment, 12–13
universal character sets, 10–3
UNIX
  IPC domain, 12–2
  IPC socket pairs, 12–18
  kernel, 2–1, 2–3, 2–10
  processes, 2–1
unix_master( ) routine, 12–21
unlinking multiplexed lower streams, 13–25
unloading a file system type, 9–8
unresolved reference, 8–1, 8–6, 8–15
untagged file systems, 11–38
unweld request, 13–8
unweldq( ) routine, 13–22
update_priority( ) routine, 5–7
usage-aging mechanism, 5–3, 5–6
user area, physical volumes, 14–8
user data area, 14–11
user ID, 2–5
user mode, 2–10
user structure, 2–5, 4–2
user structure (UNIX), 4–4
user timer, 5–26, 5–27
utask structure, 4–2, 4–4, 4–5, 4–16, 4–17, 4–18
uthread structure, 4–2, 4–4, 4–5, 4–11, 4–16, 4–17

V

VFS, 9–8, 11–1
    architecture, 11–8
    interface changes, 11–32
    layer, 11–8
    operations, 11–8
    switch, 9–8, 11–10, 11–38
vfssw_add() routine, 11–38
vfssw_del() routine, 11–38
VGDA. *See* volume group descriptor area
VGRA. *See* volume group reserved area
VGSA. *See* volume group status area
virtual
    address map, 3–3, 3–8, 3–9, 3–10, 3–16
    address space, 6–1
    memory object (VM object), 3–9, 3–11 to 3–13, 6–1 to 6–29, 7–16
Virtual File System. *See* VFS
vm_allocate() routine, 3–18
vm_copy() routine, 3–19
vm_deallocate() routine, 3–18
vm_fault() routine, 6–26
vm_inherit() routine, 6–28
vm_inheritance() routine, 3–19
vm_map structure, 6–2, 6–3, 6–4
vm_map(), 7–19
vm_map() routine, 3–18
vm_map_entry structure, 6–2, 6–4, 6–6
vm_page data structure, 3–15
vm_protect() interface, 6–6
vm_protect() routine, 3–19
vm_read() routine, 3–19
vm_region() routine, 3–19
vm_write() routine, 3–19
vnode, 11–1, 11–3
    additions, 11–40
    contents, 11–17
    covered, 11–11
    free list, 11–17, 11–19, 11–21
    interface changes, 11–32
    invalidation of, 11–24
    layer, 11–8
    list, 11–11
    management, 11–17
    operations, 11–8
    operations vector, 11–19
    pager, 3–13, 6–30, 7–3 to 7–14
    recycling, 11–16, 11–23
    types, 11–18
vnodeops structure, 15–59
vnode_pager_data_request_direct() routine, 7–12
vn_write() routine, 11–24
volume group descriptor area, 14–9, 14–10
volume group reserved area, 14–8, 14–9, 14–19
volume group status area, 14–9, 14–10, 14–13, 14–18, 14–25
volume groups, 14–4
    configuration, 14–10, 14–18, 14–25
    data structures, 14–15
volumes, 14–2, 14–4
vstruct data structure, 7–7

W

wait queue, 5–13, 5–14, 5–15, 5–16
wait() system call, 2–3, 4–1, 4–6
wakeup() routine, 12–8, 13–19, 13–23
weld request, 13–8, 13–16, 13–22
weldq() routine, 13–22
wired pages, 3–15
wired-down memory, 6–30
    and kernel stack, 7–15

Notes

Notes

Notes

OPEN SOFTWARE FOUNDATION™

INFORMATION REQUEST FORM

Please send to me the following:

( ) OSF™ Membership Information

( ) OSF/1™ License Materials

( ) OSF/1 Training Information

Contact Name ______________________________

Company Name ______________________________

Street Address ______________________________

Mail Stop ______________________________

City ______________ State ____ Zip __________

Phone ______________ FAX ______________

Electronic Mail ______________________________

MAIL TO:

Open Software Foundation
11 Cambridge Center
Cambridge, MA 02142

Attn: OSF/1

For more information about OSF/1 call OSF Direct Channels at 617 621 7300.

OSF/1™ Operating System

Release 1.2

Design of the OSF/1 Operating System

Titles in the OSF/1 Operating System Series

- Design of the OSF/1 Operating System
- OSF/1 User's Guide
- OSF/1 Command Reference
- OSF/1 Programmer's Reference
- OSF/1 System and Network Administrator's Reference
- OSF/1 Network Applications Programmer's Guide
- Application Environment Specification (AES) Operating System Programming Interfaces Volume

Printed in the U.S.A.

ISBN 0-13-202813-1

Open Software Foundation
11 Cambridge Center
Cambridge, Massachusetts 02142

Prentice-Hall, Inc.

OPERATING SYSTEMS