THE DESIGN AND IMPLEMENTATION OF THE FreeBSD® OPERATING SYSTEM

SECOND EDITION

MARSHALL KIRK McKUSICK
GEORGE V. NEVILLE-NEIL
ROBERT N.M. WATSON

Sample Pages

Contents

Preface xxi

About the Authors xxix

Part I Overview 1

Chapter 1 History and Goals 3

1.1 History of the UNIX System 3
- Origins 3
- Research UNIX 4
- AT&T UNIX System III and System V 5
- Berkeley Software Distributions 6
- UNIX in the World 7

1.2 BSD and Other Systems 7
- The Influence of the User Community 8

1.3 The Transition of BSD to Open Source 9
- Networking Release 2 10
- The Lawsuit 11
- 4.4BSD 13
- 4.4BSD-Lite Release 2 13

1.4 The FreeBSD Development Model 14
- References 17

Chapter 2 Design Overview of FreeBSD 21

2.1 FreeBSD Facilities and the Kernel 21
- The Kernel 22

2.2 Kernel Organization 23

Sample pages

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

Chapter 3 Kernel Services 57

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

Sample pages

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

Part II Processes 87

Chapter 4 Process Management 89

4.1 Introduction to Process Management 89
Multiprogramming 90
Scheduling 91
4.2 Process State 92
The Process Structure 94
The Thread Structure 98
4.3 Context Switching 99
Thread State 100
Low-Level Context Switching 100
Voluntary Context Switching 101
Synchronization 106
Mutex Synchronization 107
Mutex Interface 109
Lock Synchronization 110
Deadlock Prevention 112
4.4 Thread Scheduling 114
The Low-Level Scheduler 114
Thread Run Queues and Context Switching 115
Timeshare Thread Scheduling 117
Multiprocessor Scheduling 122
Adaptive Idle 125

Sample pages

Traditional Timeshare Thread Scheduling 125
4.5 Process Creation 126
4.6 Process Termination 128
4.7 Signals 129
Posting of a Signal 132
Delivering a Signal 135
4.8 Process Groups and Sessions 136
Process Groups 137
Sessions 138
Job Control 139
4.9 Process Debugging 142
Exercises 144
References 146

Chapter 5 Security 147

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

Sample pages

Capability Mode 179
5.9 Jails 180
5.10 Mandatory Access-Control Framework 184
Mandatory Policies 186
Guiding Design Principles 187
Architecture of the MAC Framework 188
Framework Startup 189
Policy Registration 190
Framework Entry-Point Design Considerations 191
Policy Entry-Point Considerations 192
Kernel Service Entry-Point Invocation 193
Policy Composition 194
Object Labelling 195
Label Life Cycle and Memory Management 196
Label Synchronization 199
Policy-Agnostic Label Management from Userspace 199
5.11 Security Event Auditing 200
Audit Events and Records 201
BSM Audit Records and Audit Trails 202
Kernel-Audit Implementation 203
5.12 Cryptographic Services 206
Cryptographic Framework 206
Random-Number Generator 208
5.13 GELI Full-Disk Encryption 212
Confidentiality and Integrity Protection 212
Key Management 213
Starting GELI 214
Cryptographic Block Protection 215
I/O Model 216
Limitations 216
Exercises 217
References 217

Chapter 6 Memory Management 221

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

Sample pages

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

Sample pages

Exercises 308
References 310

Part III I/O System 313

Chapter 7 I/O System Overview 315

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

Chapter 8 Devices 361

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

Sample pages

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

Chapter 9 The Fast Filesystem 431

9.1 Hierarchical Filesystem Management 431
9.2 Structure of an Inode 433
Changes to the Inode Format 435
Extended Attributes 436
New Filesystem Capabilities 438

Sample pages

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

Sample pages

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

Chapter 10 The Zettabyte Filesystem 523

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

Chapter 11 The Network Filesystem 551

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

Sample pages

Exercises 586
References 587

Part IV Interprocess Communication 591

Chapter 12 Interprocess Communication 593

12.1 Interprocess-Communication Model 593
Use of Sockets 596
12.2 Implementation Structure and Overview 599
12.3 Memory Management 601
Mbufs 601
Storage-Management Algorithms 605
Mbuf Utility Routines 606
12.4 IPC Data Structures 606
Socket Addresses 611
Locks 612
12.5 Connection Setup 612
12.6 Data Transfer 615
Transmitting Data 616
Receiving Data 617
12.7 Socket Shutdown 620
12.8 Network-Communication Protocol Internal Structure 621
Data Flow 623
Communication Protocols 624
12.9 Socket-to-Protocol Interface 626
Protocol User-Request Routines 627
Protocol Control-Output Routine 630
12.10 Protocol-to-Protocol Interface 631
pr_output 632
pr_input 632
pr_ctlinput 633
12.11 Protocol-to-Network Interface 634
Network Interfaces and Link-Layer Protocols 634
Packet Transmission 641
Packet Reception 642
12.12 Buffering and Flow Control 643
Protocol Buffering Policies 643
Queue Limiting 643
12.13 Network Virtualization 644
Exercises 646
References 648

Chapter 13 Network-Layer Protocols 649

13.1 Internet Protocol Version 4 650
IPv4 Addresses 652
Broadcast Addresses 653

Sample pages

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

Chapter 14 Transport-Layer Protocols 721

14.1 Internet Ports and Associations 721
Protocol Control Blocks 722
14.2 User Datagram Protocol (UDP) 723
Initialization 723
Output 724
Input 724
Control Operations 725
14.3 Transmission Control Protocol (TCP) 725
TCP Connection States 727

Sample pages

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

Part V System Operation 773

Chapter 15 System Startup and Shutdown 775

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

Sample pages

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

Glossary **807**

Index **847**

Sample pages

CHAPTER 4

Process Management

Sample pages

4.1 Introduction to Process Management

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

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

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

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

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

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

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

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

Sample pages

Multiprogramming

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

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

those registers with the new hardware state. All architectures must save and restore the software state used by the kernel.

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

Scheduling

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

Sample pages

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

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

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

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

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

Sample pages

4.2 Process State

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

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

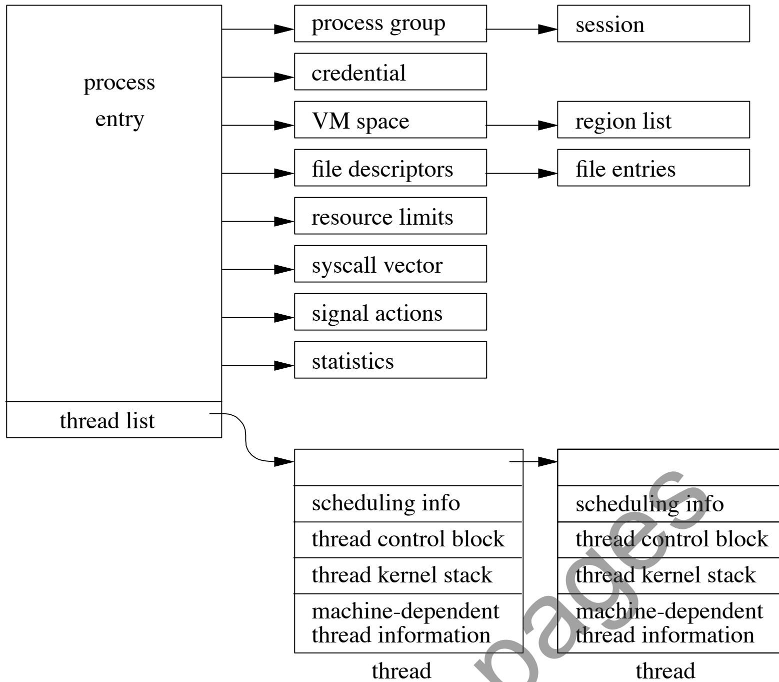


Figure 4.1 Process state.

Sample pages

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

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

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

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

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

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

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

Sample pages

The Process Structure

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

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

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

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

Sample pages

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

Table 4.1 Process states.

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

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

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

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

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

Sample pages

Figure 4.2 Process-group hierarchy.

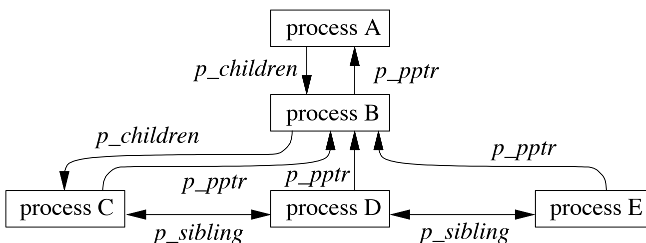


Table 4.2 Thread-scheduling classes.

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

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

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

Sample pages

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

When a thread goes to sleep in the kernel, it must specify whether it should be awakened and marked runnable if a signal is posted to it. In FreeBSD, a kernel thread will be awakened by a signal only if it sets the PCATCH flag when it sleeps.

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

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

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

Sample pages

The Thread Structure

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

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

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

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

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

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

Sample pages

4.3 Context Switching

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

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

Thread State

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

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

Sample pages

Low-Level Context Switching

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

Voluntary Context Switching

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

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

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

Sample pages

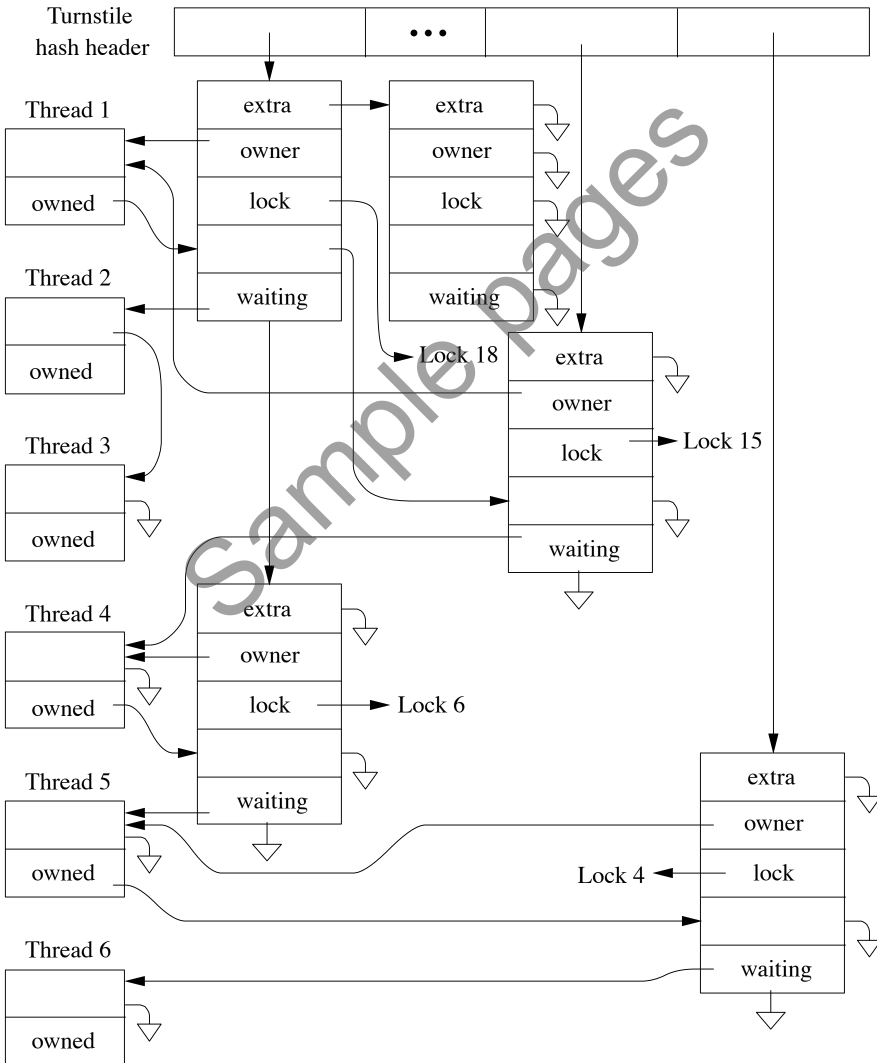
A thread may block for a short, medium, or long period of time depending on the reason that it needs to wait. A short wait occurs when a thread needs to wait for access to a lock that protects a data structure. A medium wait occurs while a thread waits for an event that is expected to occur quickly such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

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

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

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

Figure 4.3 Turnstile structures for blocked threads.



Sample pages