THE DESIGN AND IMPLEMENTATION OF THE FreeBSD® OPERATING SYSTEM

SECOND EDITION

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

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

Second Edition

This page intentionally left blank

The Design and Implementation of the FreeBSD® Operating System

Second Edition

Marshall Kirk McKusick

George V. Neville-Neil

Robert N.M. Watson

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data

McKusick, Marshall Kirk.
The design and implementation of the FreeBSD operating system / Marshall Kirk McKusick, George V. Neville-Neil, Robert N. M. Watson.
pages cm
Includes bibliographical references and index.
ISBN-13: 978-0-321-96897-5 (hardcover : alk. paper)
ISBN-10: 0-321-96897-2 (hardcover : alk. paper)
1. FreeBSD. 2. Free computer software. 3. Operating systems (Computers)
I. Neville-Neil, George V. II. Watson, Robert N. M. III. Title.
QA76.774.F74M35 2014
005.4'32—dc23 2014020072

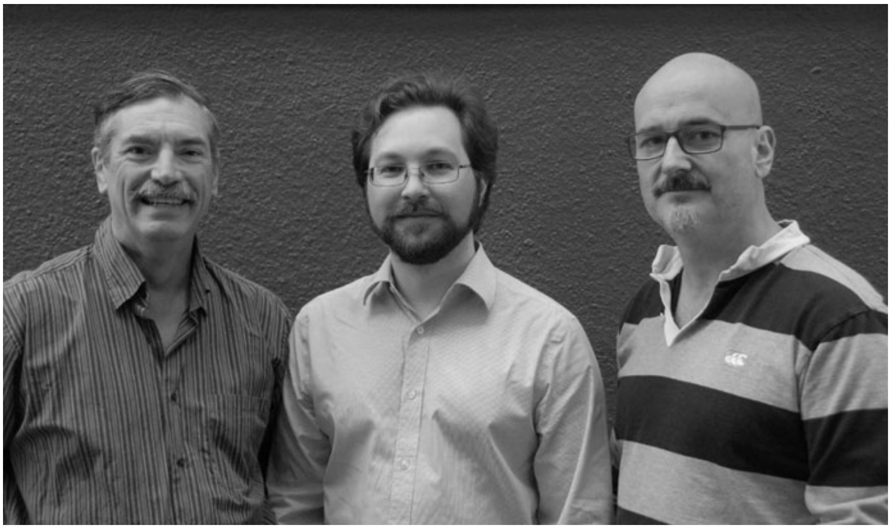
ISBN-13: 978-0-321-96897-5
ISBN-10: 0-321-96897-2
Text printed on recycled and acid-free paper at Courier in Westford, Massachusetts.
Second Printing, November 2014

Dedication

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

This page intentionally left blank

About the Authors



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

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

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

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

This page intentionally left blank

CHAPTER 4

Process Management

4.1 Introduction to Process Management

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

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

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

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

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

The resources used by a thread are split into two parts. The resources needed for execution in user mode are defined by the CPU architecture and typically include the CPU's general-purpose registers, the program counter, the 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.

Multiprogramming

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

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

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

Scheduling

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

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

Some tasks, such as the compilation of a large application, may be done in many small steps in which each component is compiled in a separate process. No individual step runs long enough to have its priority degraded, so the compilation as a whole impacts the interactive programs. To detect and avoid this problem, the scheduling priority of a child process is propagated back to its parent. When a new child process is started, it begins running with its parent's current priority. As the program that coordinates the compilation (typically **make**) starts many compilation steps, its priority is dropped because of the CPU-intensive behavior of its children. Later compilation steps started by **make** begin running and stay at a lower priority, which allows higher-priority interactive programs to run in preference to them as desired.

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

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

4.2 Process State

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

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

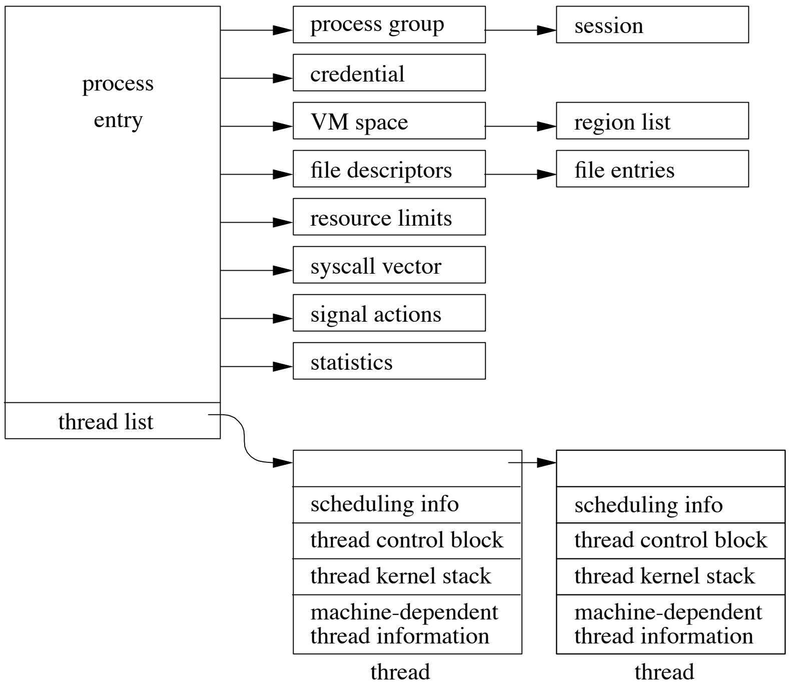


Figure 4.1 Process state.

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

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

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

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

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

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

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

The Process Structure

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

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

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

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

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

Table 4.1 Process states.

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

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

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

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

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

Figure 4.2 Process-group hierarchy.

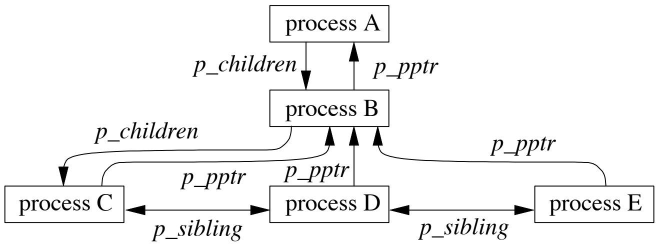


Table 4.2 Thread-scheduling classes.

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.

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

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

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

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

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

The Thread Structure

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

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

Historically, the kernel stack was mapped to a fixed location in the virtual address space. The reason for using a fixed mapping is that when a parent forks, its run-time stack is copied for its child. If the kernel stack is mapped to a fixed address, the child's kernel stack is mapped to the same addresses as its parent kernel stack. Thus, all its internal references, such as frame pointers and stack-variable references, work as expected.

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

Every thread that might potentially run must have its stack resident in memory because one task of its stack is to handle page faults. If it were not resident, it would page fault when the thread tried to run, and there would be no kernel stack available to service the page fault. Since a system may have many thousands of threads, the kernel stacks must be kept small to avoid wasting too much physical memory. In FreeBSD on the Intel architecture, the kernel stack is limited to two pages of memory. Implementors must be careful when writing code that executes in the kernel to avoid using large local variables and deeply nested subroutine calls to avoid overflowing the run-time stack. As a safety precaution, some architectures leave an invalid page between the area for the run-time stack and the data structures that follow it. Thus, overflowing the kernel stack will cause a 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.

4.3 Context Switching

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

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

Thread State

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

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

Low-Level Context Switching

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

Voluntary Context Switching

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

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

- When a parent process does a *wait* system call to collect the termination status of its children, it must wait for one of those children to exit. Since it cannot know which of its children will exit first, and since it can sleep on only a single wait channel, there is a quandary about how to wait for the next of multiple events. The solution is to have the parent sleep on its own process structure. When a child exits, it awakens its parent's process-structure address rather than its own. Thus, the parent doing the *wait* will awaken independently of which child process is the first to exit. Once running, it must scan its list of children to determine which one exited.

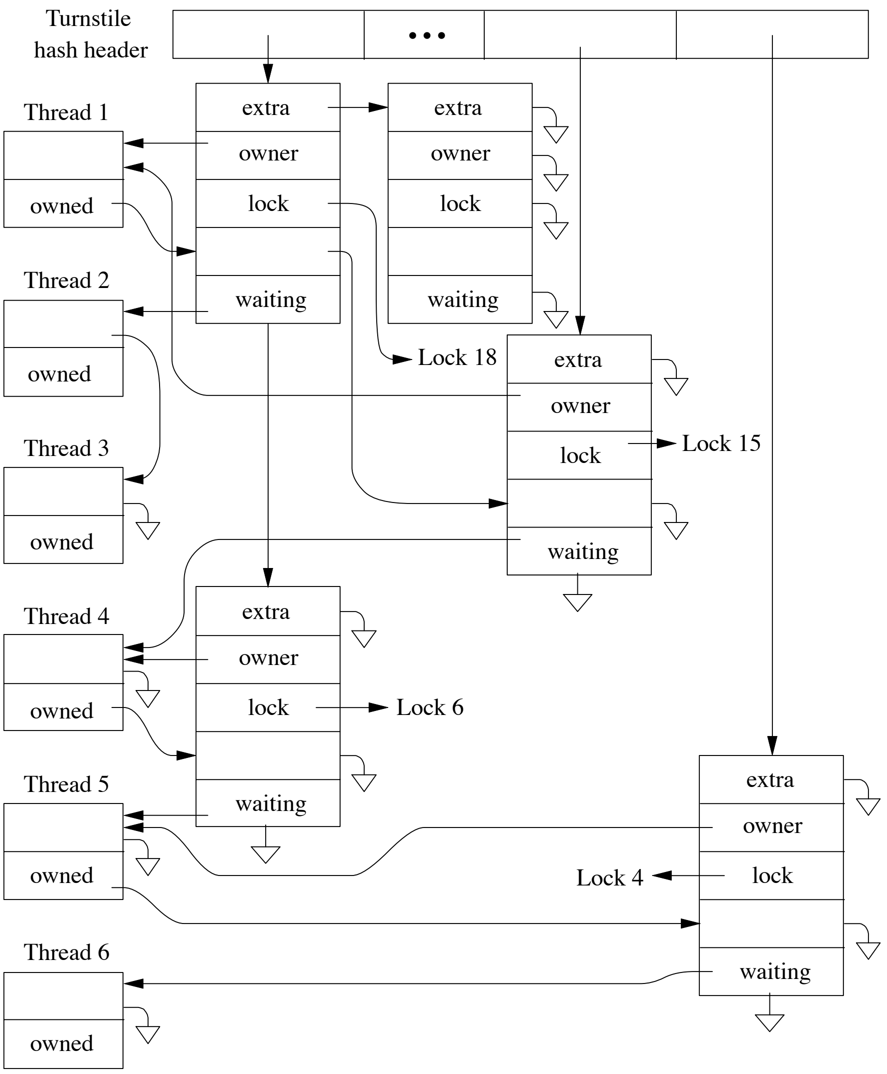
- When a thread does a *sigsuspend* system call, it does not want to run until it receives a signal. Thus, it needs to do an interruptible sleep on a wait channel that will never be awakened. By convention, the address of the signal-actions structure is given as the wait channel.

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

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

A short-term lock is managed by a ***turnstile*** data structure. The *turnstile* tracks the current owner of the lock and the list of threads waiting for access to the lock. Figure 4.3 shows how *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.



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

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

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

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

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

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

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

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

Suspending a thread takes the following steps in its operation:

1. Prevents events that might cause thread-state transitions. Historically a global scheduling lock was used, but it was a bottleneck. Now each thread uses a lock tied to its current state to protect its per-thread state. For example, when a thread is on a run queue, the lock for that run queue is used; when the thread is blocked on a *turnstile*, the *turnstile*'s lock is used; when a thread is blocked on a sleep queue, the lock for the wait channels hash chain is used.

2. Records the wait channel in the thread structure and hashes the wait-channel value to check for an existing *turnstile* or *sleepqueue* for the wait-channel. If one exists, links the thread to it and saves the *turnstile* or *sleepqueue* structure provided by the thread. Otherwise places the *turnstile* or *sleepqueue* onto the hash chain and links the thread into it.

3. For threads being placed on a *turnstile*, if the current thread's priority is higher than the priority of the thread currently holding the lock, propagates the current thread's priority to the thread currently holding the lock. For threads being placed on a *sleepqueue*, sets the thread's priority to the priority that the thread will have when the thread is awakened and sets its SLEEPING flag.

4. For threads being placed on a *turnstile*, sort the thread into the list of waiting threads such that the highest priority thread appears first in the list. For threads being placed on a *sleepqueue*, place the thread at the end of the list of threads waiting for that wait-channel.

5. Calls *mi_switch*() to request that a new thread be scheduled; the associated mutex is released as part of switching to the other thread.

A sleeping thread is not selected to execute until it is removed from a *turnstile* or *sleepqueue* and is marked runnable. This operation is done either implicitly as part of a lock being released, or explicitly by a call to the *wakeup*() routine to signal that an event has occurred or that a resource is available. When *wakeup*() is invoked, it is given a wait channel that it uses to find the corresponding *sleepqueue* (using a hashed lookup). It awakens all threads sleeping on that wait channel. All threads waiting for the resource are awakened to ensure that none are inadvertently left sleeping. If only one thread were awakened, it might not request the resource on which it was sleeping. If it does not use and release the resource, any other threads waiting for that resource will be left sleeping forever. A thread that needs an empty disk buffer in which to write data is an example of a thread that may not request the resource on which it was sleeping. Such a thread can use any available buffer. If none is available, it will try to create one by requesting that a dirty buffer be written to disk and then waiting for the I/O to complete. When the I/O finishes, the thread will awaken and will check for an empty buffer. If several are available, it may not use the one that it cleaned, leaving any other threads to sleep forever as they wait for the cleaned buffer.

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

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

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

Resuming a thread takes the following steps in its operation:

1. Removes the thread from its *turnstile* or *sleepqueue*. If it is the last thread to be awakened, the *turnstile* or *sleepqueue* is returned to it. If it is not the last thread to be awakened, a *turnstile* or *sleepqueue* from the free list is returned to it.

2. Recomputes the user-mode scheduling priority if the thread has been sleeping longer than one second.

3. If the thread had been blocked on a *turnstile*, it is placed on the run queue. If the thread had been blocked on a *sleepqueue*, it is placed on the run queue if it is in a SLEEPING state and if its process is not swapped out of main memory.

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

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

Synchronization

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

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

Table 4.3 Locking hierarchy.

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

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

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

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

Mutex Synchronization

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

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

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

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

Mutexes that do not sleep are called ***spin mutexes***. A spin mutex will not relinquish the CPU when it cannot immediately get the requested lock, but it will loop, waiting for the mutex to be released by another CPU. Spinning can result in deadlock if a thread interrupted the thread that held a mutex and then tried to acquire the mutex. To protect an interrupt thread from blocking against itself during the period that it is held, a spin mutex runs inside a critical section with interrupts disabled on that CPU. Thus, an interrupt thread can run only on another CPU during the period that the spin mutex is held.

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

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

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

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

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

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

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

Mutex Interface

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

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

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

The *mtx_trylock*() function tries to acquire a mutual exclusion lock for the currently running kernel thread. If the mutex cannot be immediately acquired, *mtx_trylock*() will return 0; otherwise the mutex will be acquired and a nonzero value will be returned. The *mtx_trylock*() function cannot be used with spin mutexes.

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

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

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

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

Lock Synchronization

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

- Request shared: Get one of many possible shared locks. If a thread holding an exclusive lock requests a shared lock, some lock managers will downgrade the exclusive lock to a shared lock while others simply return an error.

- Request exclusive: When all shared locks have cleared, grant an exclusive lock. To ensure that the exclusive lock will be granted quickly, some lock managers stop granting shared locks when an exclusive lock is requested. Others grant new shared locks only for recursive lock requests. Only one exclusive lock may exist at a time, except that a thread holding an exclusive lock may get additional exclusive locks if the *canrecurse* flag was set when the lock was initialized. Some lock managers allow the *canrecurse* flag to be specified in the lock request.

- Request release: Release one instance of a lock.

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

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

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

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

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

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

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

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

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

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

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

Deadlock Prevention

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

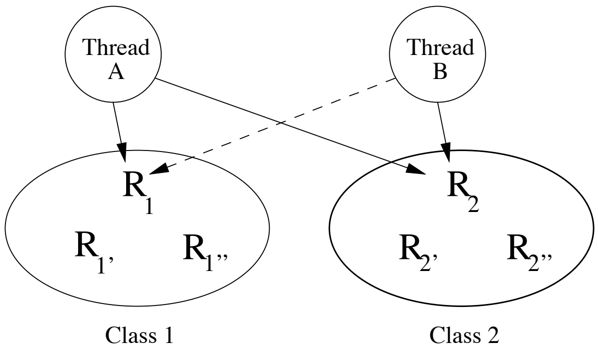


Figure 4.4 Partial ordering of resources.

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

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

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

Historically, the class members and ordering were poorly documented and unenforced. Violations were discovered when threads would deadlock and a careful analysis was done to figure out what ordering had been violated. With an increasing number of developers and a growing kernel, the ad hoc method of maintaining the partial ordering of locks became untenable. A witness module was added to the kernel to derive and enforce the partial ordering of the locks. The witness module keeps track of the locks acquired and released by each thread. It also keeps track of the order in which locks are acquired relative to each other. Each time a lock is acquired, the witness module uses these two lists to verify that a lock is not being acquired in the wrong order. If a lock order violation is detected, then a message is output to the console detailing the locks involved and the locations in the code in which they were acquired. The witness module also verifies that no locks that prohibit sleeping are held when requesting a sleep lock or voluntarily going to sleep.

The witness module can be configured to either panic or drop into the kernel debugger when an order violation occurs or some other witness check fails. When running the debugger, the witness module can output the list of locks held by the current thread to the console along with the filename and line number at which each lock was last acquired. It can also dump the current order list to the console. The code first displays the lock order tree for all the sleep locks. Then it displays the lock order tree for all the spin mutexes. Finally, it displays a list of locks that have not yet been acquired.

4.4 Thread Scheduling

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

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

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

The Low-Level Scheduler

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

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

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

Thread Run Queues and Context Switching

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

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

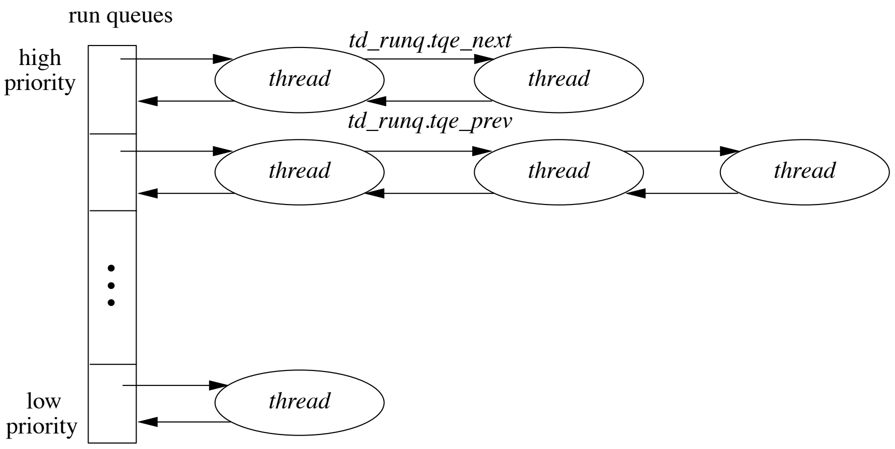


Figure 4.5 Queueing structure for runnable threads.

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

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

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

This mechanism is handled by setting the currently running thread's TDF_NEEDRESCHED flag and then posting an ***asynchronous system trap*** (***AST***). An AST is a trap that is delivered to a thread the next time that thread is preparing to return from an interrupt, a trap, or a system call. Some architectures support ASTs directly in hardware; other systems emulate ASTs by checking an AST flag at the end of every system call, trap, and interrupt. When the hardware AST trap occurs or the AST flag is set, the *mi_switch*() routine is called instead of the current thread resuming execution. Rescheduling requests are made by the *sched_lend_user_prio*(), *sched_clock*(), *sched_setpreempt*(), and *sched_affinity*() routines.

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

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

Timeshare Thread Scheduling

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

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

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

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

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

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

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

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

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

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

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

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

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

A particular challenge is complex X Window applications such as Web browsers and office productivity packages. These applications may consume significant resources for brief periods of time, however the user expects them to remain interactive. To resolve this issue, a several-second history of the sleep and run behavior is kept and gradually decayed. Thus, the scheduler keeps a moving average that can tolerate bursts of behavior but will quickly penalize timeshare threads that abuse their elevated status. A longer history allows longer bursts but learns more slowly.

The interactivity score is compared to the interactivity threshold, which is the cutoff point for considering a thread interactive. The interactivity threshold is modified by the process ***nice*** value. Positive nice values make it more challenging for a thread to be considered interactive, while negative values make it easier. Thus, the nice value gives the user some control over the primary mechanism of reducing thread-scheduling latency.

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

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

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

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

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

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

The CPU utilization estimator accumulates run time as a thread runs and decays it as a thread sleeps. The utilization estimator provides the percent-cpu values displayed in **top** and **ps**. ULE delays the decay until a thread wakes to avoid periodically scanning every thread in the system. Since this delay leaves values unchanged for the duration of sleeps, the values must also be decayed before any user process inspects them. This approach preserves the constant-time and event-driven nature of the scheduler.

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

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

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

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

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

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

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

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

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

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

Multiprocessor Scheduling

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

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

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

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

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

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

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

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

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

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

The long-term load balancer balances the worst path pair in the hierarchy to avoid socket-, cache-, and chip-level imbalances. It runs from an interrupt handler in a randomized interval of roughly 1 second. The interval is randomized to prevent harmonic relationships between periodic threads and the periodic load balancer. In much the same way a stochastic sampling profiler works, the balancer picks the most- and least-loaded path from the current tree position and then recursively balances those paths by migrating threads.

The scheduler must decide whether it is necessary to send an IPI when adding a thread to a remote CPU, just as it must decide whether adding a thread to the current CPU should preempt the current thread. The decision is made based on the current priority of the thread running on the target CPU and the priority of the thread being scheduled. Preempting whenever the pushed thread has a higher priority than the currently running thread results in excessive interrupts and preemptions. Thus, a thread must exceed the timesharing priority before an IPI is generated. This requirement trades some latency in batch jobs for improved performance.

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

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

Adaptive Idle

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

Traditional Timeshare Thread Scheduling

The traditional FreeBSD timeshare-scheduling algorithm is based on ***multilevel feedback queues***. The system adjusts the priority of a thread dynamically to reflect resource requirements (e.g., being blocked awaiting an event) and the amount of resources consumed by the thread (e.g., CPU time). Threads are moved between run queues based on changes in their scheduling priority (hence the word "feedback" in the name ***multilevel feedback queue***). When a thread other than the currently running thread attains a higher priority (by having that priority either assigned or given when it is awakened), the system switches to that thread immediately if the current thread is in user mode. Otherwise, the system switches to the higher-priority thread as soon as the current thread exits the kernel. The system tailors this ***short-term-scheduling algorithm*** to favor interactive jobs by raising the scheduling priority of threads that are blocked waiting for I/O for 1 or more seconds and by lowering the priority of threads that accumulate significant amounts of CPU time.

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

4.5 Process Creation

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

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

A *fork* involves three main steps:

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

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

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

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

The child's explicitly set information includes:

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

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

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

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

4.6 Process Termination

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

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

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

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

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

With the kernel-mode state reset, the process is then removed from the list of active processes—the *allproc* list—and is placed on the list of ***zombie processes*** pointed to by *zombproc*. The process state is changed to show that no thread is currently running. The *exit*() routine then does the following:

- Records the termination status in the *p_xstat* field of the process structure
- Bundles up a copy of the process's accumulated resource usage (for accounting purposes) and hangs this structure from the *p_ru* field of the process structure
- Notifies the deceased process's parent

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

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

4.7 Signals

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

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

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

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

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

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

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

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

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

Table 4.5 Signals defined in FreeBSD.

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

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

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

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

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

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

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

Posting of a Signal

The implementation of signals is broken up into two parts: posting a signal to a process and recognizing the signal and delivering it to the target thread. Signals may be posted by any process or by code that executes at interrupt level. Signal delivery normally takes place within the context of the receiving thread. When a signal forces a process to be stopped, the action can be carried out on all the threads associated with that process when the signal is posted.

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

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

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

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

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

2. Given an action, *psignal*() selects the appropriate thread and adds the signal to the thread's set of pending signals, *td_siglist*, and then does any implicit actions specific to that signal. For example, if the signal is the continue signal, SIGCONT, any pending signals that would normally cause the process to stop, such as SIGTTOU, are removed.
3. Next, *psignal*() checks whether the signal is being masked. If the thread is currently masking delivery of the signal, *psignal*()'s work is complete and it may return.
4. If the signal is not being masked, *psignal*() must either perform the action directly or arrange for the thread to execute so that the thread will take the action associated with the signal. Before setting the thread to a runnable state, *psignal*() must take different courses of action depending on the state of the thread as follows:

SLEEPING The thread is blocked awaiting an event. If the thread is sleeping noninterruptibly, then nothing further can be done. Otherwise, the kernel can apply the action—either directly or indirectly—by waking up the thread. There are two actions that can be applied directly. For signals that cause a process to stop, all the threads in the process are placed in the STOPPED state, and the parent process is notified of the state change by a SIGCHLD signal being posted to it. For signals that are ignored by default, the signal is removed from the signal list and the work is complete. Otherwise, the action associated with the signal must be done in the context of the receiving thread, and the thread is placed onto the run queue with a call to *setrunnable*().

STOPPED The process is stopped by a signal or because it is being debugged. If the process is being debugged, then there is nothing to do until the controlling process permits it to run again. If the process is stopped by a signal and the posted signal would cause the process to stop again, then there is nothing to do, and the posted signal is discarded. Otherwise, the signal is either a continue signal or a signal that would normally cause the process to terminate (unless the signal is caught). If the signal is SIGCONT, then all the threads in the process that were previously running are set running again. Any threads in the process that were blocked waiting on an event are returned to the SLEEPING state. If the signal is SIGKILL, then all the threads in the process are set running again no matter what, so that they can terminate the next time that they are scheduled to run. Otherwise, the signal causes the threads in the process to be made runnable, but the threads are not placed on the run queue because they must wait for a continue signal.

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

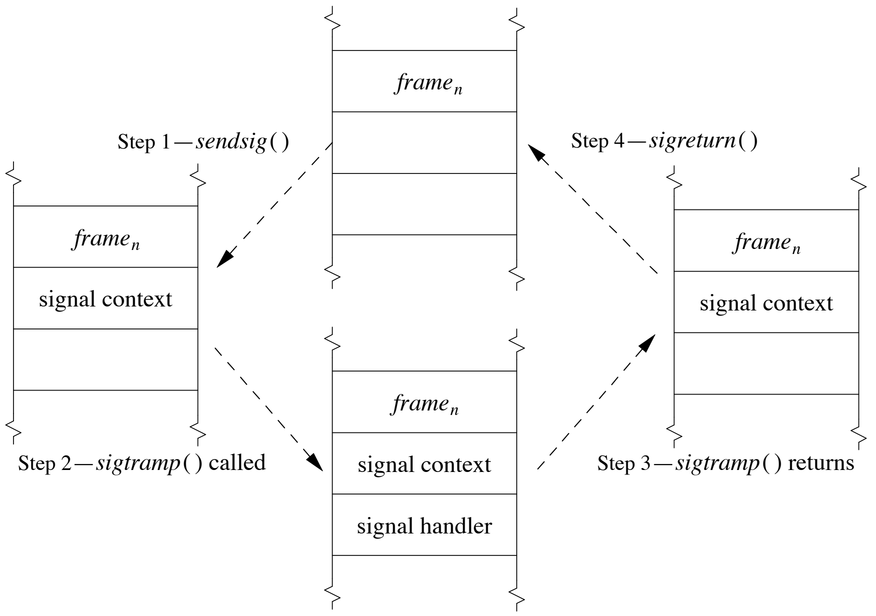


Figure 4.6 Delivery of a signal to a process. Step 1: The kernel places a signal context on the user's stack. Step 2: The kernel places a signal-handler frame on the user's stack and arranges to start running the user process in the *sigtramp*() code. When the *sigtramp*() routine starts running, it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

Delivering a Signal

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

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

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

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

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

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

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

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

4.8 Process Groups and Sessions

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

Sessions were designed by the IEEE POSIX.1003.1 Working Group with the intent of fixing a long-standing security problem in UNIX—namely, that processes could modify the state of terminals that were trusted by another user's processes. A ***session*** is a collection of process groups, and all members of a process group are members of the same session. In FreeBSD, when a user first logs onto the system, he is entered into a new session. Each session has a ***controlling process***, which is normally the user's login shell. All subsequent processes created by the user are part of process groups within this session, unless he explicitly creates a new session. Each session also has an associated login name, which is usually the user's login name. This name can be changed by only the superuser.

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

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

Process Groups

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

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

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

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

Sessions

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

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

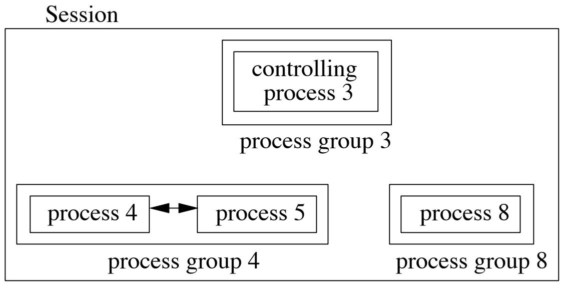
A login session is created by a program that prepares a terminal for a user to log into the system. That process normally executes a shell for the user, and thus the shell is created as the controlling process. An example of a typical login session is shown in Figure 4.7.

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

Job Control

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

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



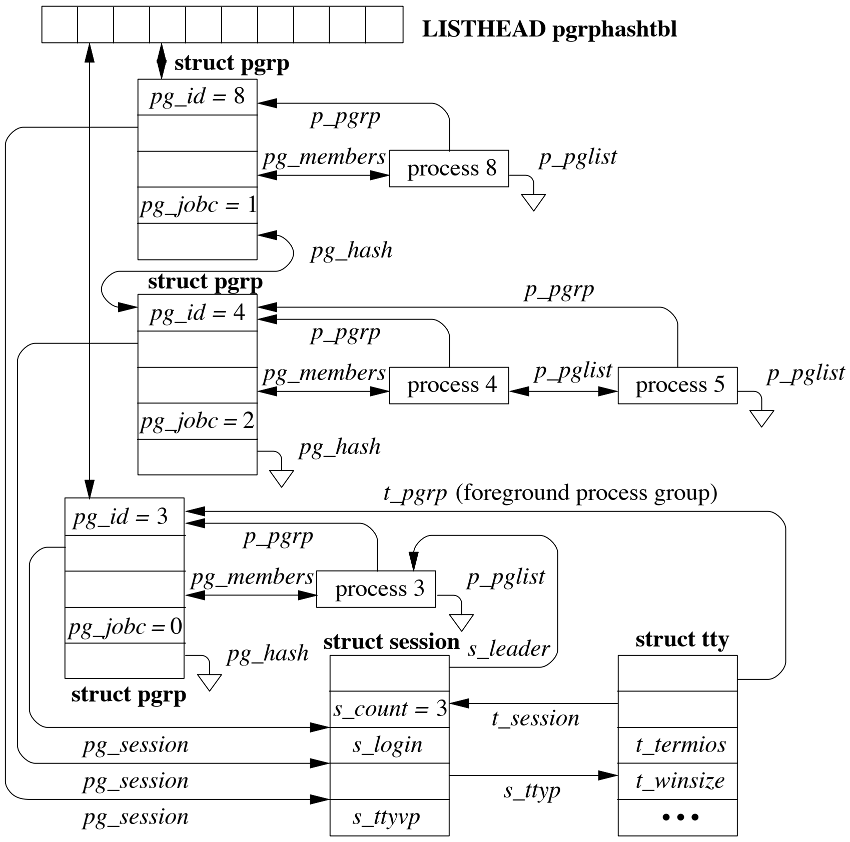


Figure 4.8 Process-group organization.

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

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

For example, special characters typed at the keyboard of the terminal (e.g., control-C or control-\) result in a signal being sent to all processes in one job in the session; that job is in the ***foreground***, whereas all other jobs in the session are in the ***background***. A shell may change the foreground job by using the *tcsetpgrp*() function, implemented by the TIOCSPGRP *ioctl* on the controlling terminal. Background jobs will be sent the SIGTTIN signal if they attempt to read from the terminal, normally stopping the job. The SIGTTOU signal is sent to background jobs that attempt an *ioctl* system call that would alter the state of the terminal. The SIGTTOU signal is also sent if the TOSTOP option is set for the terminal, and an attempt is made to write to the terminal.

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

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

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

4.9 Process Debugging

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

• Attaches to an existing process to begin debugging it

• Reads and writes address space and registers

• Intercepts signals posted to the process

• Single steps and continues the execution of the process

• Terminates the execution of the process

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

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

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

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

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

Each node is a directory that contains the following entries:

ctl A write-only file that supports a variety of control operations. Control commands are written as strings to the **ctl** file. The control commands are:

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

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

dbregs Sets the debug registers as defined by the machine architecture.

etype The type of the executable referenced by the **file** entry.

file A reference to the vnode from which the process text was read. This entry can be used to gain access to the symbol table for the process or to start another copy of the process.

fpregs The floating point registers as defined by the machine architecture. It is only implemented on machines that have distinct general-purpose and floating-point register sets.

map A map of the process's virtual memory.

mem The complete virtual memory image of the process. Only those addresses that exist in the process can be accessed. Reads and writes to this file modify the process. Writes to the text segment remain private to the process. Because the address space of another process can be accessed with *read* and *write* system calls, a debugger can access a process being debugged with much greater efficiency than it can with the *ptrace* system call. The pages of interest in the process being debugged are mapped into the kernel address space. The data requested by the debugger can then be copied directly from the kernel to the debugger's address space.

regs Allows read and write access to the register set of the process.

rlimit A read-only file containing the process's current and maximum limits.

status The process status. This file is read-only and returns a single line containing multiple space-separated fields that include the command name, the process id, the parent process id, the process group id, the session id, the controlling terminal (if any), a list of the process flags, the process start time, user and system times, the wait channel message, and the process credentials.

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

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

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

Exercises

4.1 For each state listed in Table 4.1, list the system queues on which a process in that state might be found.

4.2 Why is the performance of the context-switching mechanism critical to the performance of a highly multiprogrammed system?

4.3 What effect would increasing the time quantum have on the system's interactive response and total throughput?

4.4 What effect would reducing the number of run queues from 64 to 32 have on the scheduling overhead and on system performance?

4.5 Give three reasons for the system to select a new process to run.

4.6 Describe the three types of scheduling policies provided by FreeBSD.

4.7 What type of jobs does the timeshare scheduling policy favor? Propose an algorithm for identifying these favored jobs.

4.8 When and how does thread scheduling interact with memory-management facilities?

4.9 After a process has exited, it may enter the state of being a ZOMBIE before disappearing from the system entirely. What is the purpose of the ZOMBIE state? What event causes a process to exit from ZOMBIE?

4.10 Suppose that the data structures shown in Table 4.3 do not exist. Instead, assume that each process entry has only its own PID and the PID of its parent. Compare the costs in space and time to support each of the following operations:

a. Creation of a new process

b. Lookup of the process's parent

c. Lookup of all of a process's siblings

d. Lookup of all of a process's descendants

e. Destruction of a process

4.11 What are the differences between a mutex and a lock-manager lock?

4.12 Give an example of where a mutex lock should be used. Give an example of where a lock-manager lock should be used.

4.13 A process blocked without setting the PCATCH flag may never be awakened by a signal. Describe two problems a noninterruptible sleep may cause if a disk becomes unavailable while the system is running.

4.14 Describe the limitations a jail puts on the filesystem namespace, network access, and processes running in the jail.

*4.15 In FreeBSD, the signal SIGTSTP is delivered to a process when a user types a "suspend character." Why would a process want to catch this signal before it is stopped?

*4.16 Before the FreeBSD signal mechanism was added, signal handlers to catch the SIGTSTP signal were written as

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

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

*4.17 The process-priority calculations and accounting statistics are all based on sampled data. Describe hardware support that would permit more accurate statistics and priority calculations.

*4.18 Why are signals a poor interprocess-communication facility?

**4.19 A kernel-stack-invalid trap occurs when an invalid value for the kernel-mode stack pointer is detected by the hardware. How might the system gracefully terminate a process that receives such a trap while executing on its kernel-run-time stack?

**4.20 Describe alternatives to the `test-and-set` instruction that would allow you to build a synchronization mechanism for a multiprocessor FreeBSD system.

**4.21 A lightweight process is a thread of execution that operates within the context of a normal FreeBSD process. Multiple lightweight processes may exist in a single FreeBSD process and share memory, but each is able to do blocking operations, such as system calls. Describe how lightweight processes might be implemented entirely in user mode.

References

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

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

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

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

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

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

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