CHAPTER 4

Process Management

4.1 Introduction to Process Management

A *process* is a program in execution. 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 processes may really execute concurrently. This chapter describes the composition of a process, the method that the system uses to switch between processes, and the scheduling policy that it uses to promote sharing of the CPU. It also introduces process creation and termination, and details the signal facilities 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].

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

The resources used by a process are similarly split into two parts. The resources needed for execution in user mode are defined by the CPU architecture and typically include the CPU's general-purpose registers, the program counter, the processor-status register, and the stack-related registers, as well as the contents of the memory segments that constitute the FreeBSD 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 stack pointer—and also by the state required for the FreeBSD kernel to provide system services for a process. This *kernel state* includes parameters to the current system call, the current process's user identity, scheduling information, and so on. 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 *user 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 user structure contains information that needs to be resident only when the process is executing (although user structures of other processes also may be resident). User structures are allocated dynamically through the memory-management facilities. Historically, more than one-half of the process state was stored in the user structure. In FreeBSD, the user structure is used for only a couple of structures that are referenced from the process structure. Process structures are allocated dynamically as part of process creation and are freed as part of process exit.

Multiprogramming

The FreeBSD system 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 processes. A mechanism is also provided for *scheduling* the execution of processes—that is, for deciding which one to execute next. Facilities are provided for ensuring consistent access to data structures that are shared among processes.

Context switching is a hardware-dependent operation whose implementation is influenced by the underlying hardware facilities. Some architectures provide machine instructions that save and restore the hardware-execution context of the process, including the virtual-address space. On the 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 process, reducing the amount of the information required for that context is an effective way to produce faster context switches.

Scheduling

Fair scheduling of 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 time-share 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, and so on. Some tasks require more precise control over process execution called real-time scheduling, which must ensure that processes finish computing their results by a specified deadline or in a particular order. The FreeBSD kernel implements real-time scheduling with a separate queue from the queue used for regular time-shared processes. A process with a real-time priority is not subject to priority degradation and will only be preempted by another process of equal or higher real-time priority. The FreeBSD kernel also implements a queue of processes running at idle priority. A process with an idle priority will run only when no other process in either the real-time or time-share-scheduled queues is runnable and then only if its idle priority is equal or greater than all other runnable idle priority processes.

The FreeBSD time-share 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 to each process a high execution priority and allows that process to execute for a fixed *time slice*. Processes that execute for the duration of their slice have their priority lowered, whereas processes that give up the CPU (usually because they do I/O) are allowed to remain at their priority. Processes that are inactive have their priority raised. Thus, jobs that use large amounts of CPU time sink rapidly to a low priority, whereas interactive jobs that are mostly inactive remain at a high priority so that, when they are ready to run, they will preempt the long-running lower-priority jobs. An interactive job, such as a text editor searching for a string, may become compute-bound briefly and 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 parents current priority. Thus, 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 few free memory pages 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. This marking allows 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 user structure of the marked process. The effect of these actions is to cause the process to be swapped out (see Section 5.12). 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 for being 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 are sending a signal to a process and when receiving the exit status from a deceased process. Two PIDs are of special importance to each process: the PID of the process itself and the PID of the process's parent process.

The layout of process state was completely reorganized in FreeBSD 5.2. The goal was to support multiple *thread*s that share an address space and other resources. Threads have also been called *lightweight processes* in other systems. A thread is the unit of execution of a process; it requires an address space and other resources, but it can share many of those resources with other threads. Threads sharing an address space and other resources are scheduled independently and can all do system calls simultaneously. The reorganization of process state in FreeBSD 5.2 was designed to support threads that can select the set of resources to be shared, known as *variable-weight processes* [Aral et al., 1989].

The developers did the reorganization by moving many components of process state from the process and user structures into separate substructures for each type of state information, as shown in Figure 4.1. The process structure references

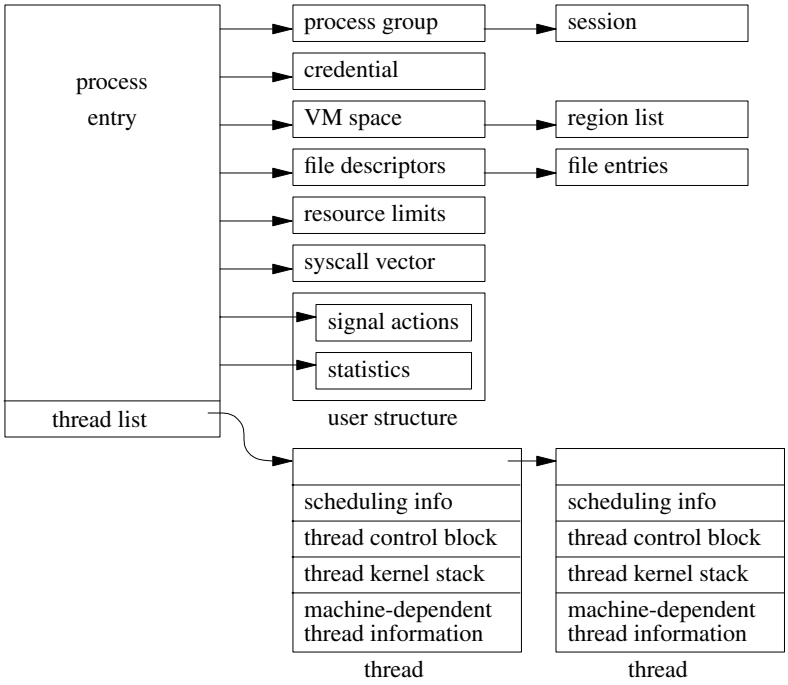


Figure 4.1 Process state.

all the substructures directly or indirectly. The user structure remains primarily as a historic artifact for the benefit of debuggers. 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 control block* (*TCB*), and other machine-dependent state. The TCB is defined by the machine architecture; it includes the general-purpose registers, stack pointers, program counter, processor-status longword, and memory-management registers.

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 *kse_create* system call. All the scheduling and management of the threads is handled by a user-level scheduler that is notified of thread transitions via callbacks from the kernel. The user-level thread manager must also 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.

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

kse_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* have just a single thread structure associated with them.

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, signal mask, 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 credentials: the real, effective, and saved user and group identifiers
- Memory management: the structure that describes the allocation of virtual address space used by the process; the VM space and its related structures are described more fully in Chapter 5
- File descriptors: an array of pointers to file entries indexed by the process 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, OSF/1, 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 3.8)
- 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)

Table 4.1 Process states.

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

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 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 fluctuate among NORMAL (where its thread(s) will be either RUNNABLE—that is, preparing to be or actually executing; SLEEPING—that is, waiting for an event; or STOPPED—that is, stopped by a signal or the parent process) until the process terminates. 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 two queues: a *run queue* or a *sleep queue*. Threads that are in a runnable state are placed on a run queue, whereas threads that are blocked awaiting an event are located on a sleep queue. Stopped threads awaiting an event are located on a sleep queue, or they are on neither type of queue. The run queues are organized according to thread-scheduling priority and are described in Section 4.4. The sleep queues are organized in a data structure that is hashed by event identifier. This organization optimizes finding the sleeping threads that need to be awakened when a wakeup occurs for an event. The sleep 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 (on page 86). 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 siblings of one another. Process B typically would be a shell that

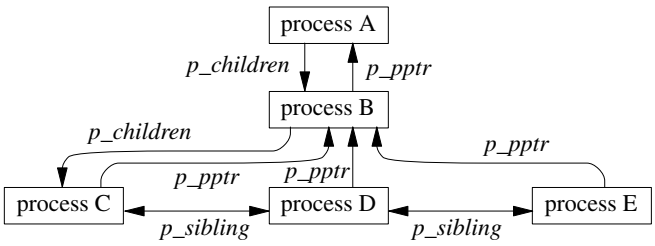


Figure 4.2 Process-group hierarchy.

started a pipeline (see Sections 2.4 and 2.6) including processes C, D, and E. Process A probably would be the system-initialization process **init** (see Sections 3.1 and 14.6).

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 top-half-kernel threads. Any runnable real-time threads are run in preference to runnable threads in the share and idle classes. Runnable time-share 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 time-share 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 *kg_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 128 to 255; priorities less than 128 are used only when a thread is *asleep*—that is, awaiting an event in the kernel—and immediately after such a thread is awakened. Threads 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

Table 4.2 Thread-scheduling classes.

Range	Class	Thread type
0 – 63	ITHD	Bottom-half kernel (interrupt)
64 – 127	KERN	Top-half kernel
128 – 159	REALTIME	Real-time user
160 – 223	TIMESHARE	Time-sharing user
224 – 255	IDLE	Idle user

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 be prepared 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 their 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 *msleep*(), 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 the *msleep*() interface 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

- Thread state: 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
- Machine state: the machine-dependent thread information
- TCB: the user- and kernel-mode execution states
- Kernel stack: the per-thread execution stack for the kernel

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 user structure to a fixed address is slow and inconvenient. FreeBSD 5.2 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 5.2 on the PC, 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 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. Our discussion will focus on how asynchronous event handling relates to synchronizing access to kernel data structures.

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 a thread's user-mode state is located in one data structure: the thread structure (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 TCB 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 TCB 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 the context of a process in the latter's thread structure permits the kernel to do context switching simply by changing the notion of the current thread structure and (if necessary) process structure, and restoring the context described by the TCB 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 process's context from its TCB. Once *mi_switch*() has loaded the execution state of the new thread, it must also check the state of the new thread for a nonlocal return request (such as when a process first starts execution after a *fork*; see Section 4.5).

Voluntary Context Switching

A *voluntary* context switch occurs whenever a thread must await the availability of a resource or the arrival of an event. Voluntary context switches happen frequently in normal system operation. For example, a thread typically blocks each time that it requests data from an input device, such as a terminal or a disk. In FreeBSD, voluntary context switches are initiated through the *sleep*() routine. When a thread no longer needs the CPU, it invokes *sleep*() with a scheduling priority and a *wait channel*. The priority specified in a *sleep*() call is the priority that should be assigned to the thread when that thread is awakened. This priority does not affect the user-level scheduling priority.

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:

- The global variable *lbolt* is awakened by the scheduler once per second. Threads that want to wait for up to 1 second can sleep on this global variable. For example, the terminal-output routines sleep on *lbolt* while waiting for output-queue space to become available. Because queue space rarely runs out, it is easier simply to check for queue space once per second during the brief periods of shortages than it is to set up a notification mechanism such as that used for managing disk buffers. Programmers can also use the *lbolt* wait channel as a crude watchdog timer when doing debugging.
- 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 on 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

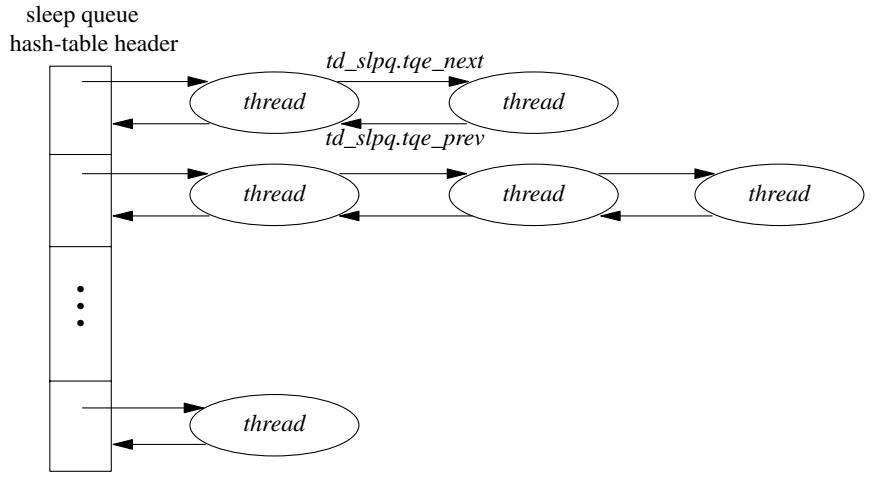
exits, it awakens its parent's process-structure address rather than its own. Thus, the parent doing the *wait* will awaken independent of which child process is the first to exit. Once running, it must scan its list of children to determine which one exited.

- When a thread does a *sigpause* 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 user structure is given as the wait channel.

Sleeping threads are organized in an array of queues (see Figure 4.3). The *sleep*( ) and *wakeup*( ) routines hash wait channels to calculate an index into the sleep queues. The *sleep*( ) routine takes the following steps in its operation:

1. Prevent interrupts that might cause thread-state transitions by acquiring the *sched_lock* mutex (mutexes are explained in the next section).
2. Record the wait channel in the thread structure and hash the wait-channel value to locate a sleep queue for the thread.
3. Set the thread's priority to the priority that the thread will have when the thread is awakened and set the SLEEPING flag.
4. Place the thread at the *end* of the sleep queue selected in step 2.
5. Call *mi_switch*( ) to request that a new thread be scheduled; the *sched_lock* mutex is released as part of switching to the other thread.

Figure 4.3 Queueing structure for sleeping threads.



A sleeping thread is not selected to execute until it is removed from a sleep queue and is marked runnable. This operation is done by the *wakeup*( ) routine, which is called to signal that an event has occurred or that a resource is available. *Wakeup*( ) is invoked with a wait channel, and it awakens *all* 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, and any other threads waiting for that resource would 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 waiting for the buffer that it cleaned sleeping forever.

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. 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.

To avoid having excessive numbers of threads awakened, kernel programmers try to use wait channels with fine enough granularity that unrelated uses will not collide on the same resource. Thus, they put locks on each buffer in the buffer cache rather than putting a single lock on the buffer cache as a whole. The problem of many threads awakening for a single resource is further mitigated on a uniprocessor by the latter's inherently single-threaded operation. Although many threads will be put into the run queue at once, only one at a time can execute. Since the uniprocessor kernel runs nonpreemptively, each thread will run its system call to completion before the next one will get a chance to execute. Unless the previous user of the resource blocked in the kernel while trying to use the resource, each thread waiting for the resource will be able to get and use the resource when it is next run.

A *wakeup*( ) operation processes entries on a sleep queue from *front* to *back*. For each thread that needs to be awakened, *wakeup*( ) does the following:

1. Removes the thread from the sleep queue
2. Recomputes the user-mode scheduling priority if the thread has been sleeping longer than one second
3. Makes the thread runnable if it is in a SLEEPING state and places the thread on the run queue 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 5.12); if the thread is in a STOPPED state, it is

Table 4.3 Locking hierarchy.

Level	Type	Sleep	Description
Lowest	hardware	no	memory-interlocked test-and-set
	spin mutex	no	spin lock
	sleep mutex	no	spin for a while, then sleep
	lock manager	yes	sleep lock
Highest	witness	yes	partially ordered sleep locks

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 *wakeup*( ) moved any threads to the run queue and one of them had 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

Historically, BSD systems ran only on uniprocessor architectures. Once a process began running in the top half of the kernel, it would run to completion or until it needed to sleep waiting for a resource to become available. The only contention for data structure access occurred at the points at which it slept or during an interrupt that needed to update a shared data structure. Synchronization with other processes was handled by ensuring that all shared data structures were consistent before going to sleep. Synchronization with interrupts was handled by raising the processor priority level to guard against interrupt activity while the shared data structure was manipulated.

Simple multiprocessor support was added to FreeBSD 3.0. It worked by creating a giant lock that protected the kernel. When a process entered the kernel it would have to acquire the giant lock before it could proceed. Thus, at most one processor could run in the kernel at a time. All the other processors could run only processes executing in user mode.

Symmetric multiprocessing (SMP) first appeared in FreeBSD 5.0 and required the addition of new synchronization schemes to eliminate the uniprocessor assumptions implicit in the FreeBSD 4.0 kernel [Schimmel, 1994]. Some subsystems in the FreeBSD 5.2 kernel have not yet been converted to symmetric multiprocessing and are still protected by the giant lock. However, most of the heavily used parts of the kernel have been moved out from under the giant lock, including much of the virtual memory system, the networking stack, and the filesystem.

Table 4.3 shows the hierarchy of locking that is necessary to support symmetric multiprocessing. The column labeled sleep in Table 4.3 shows whether a

lock of that type may be held when a thread goes to sleep. At the lowest level, the hardware must provide a memory interlocked test-and-set instruction. The test-and-set instruction must allow two operations to be done on a main-memory location—the reading of the existing value followed by the writing of a new value—without any other processor being able to read or write that memory location between the two memory operations. Some architectures support more complex versions of the test-and-set instruction. For example, the PC provides a memory interlocked compare-and-swap instruction.

Spin locks are built from the hardware test-and-set instruction. A memory location is reserved for the lock with a value of zero showing that the lock is free and a value of one showing that the lock is held. The test-and-set instruction tries to acquire the lock. The lock value is tested and the lock unconditionally set to one. If the tested value is zero, then the lock was successfully acquired and the thread may proceed. If the value is one, then some other thread held the lock so the thread must loop doing the test-and-set until the thread holding the lock (and running on a different processor) stores a zero into the lock to show that it is done with it. Spin locks are used for locks that are held only briefly—for example, to protect a list while adding or removing an entry from it.

It is wasteful of CPU cycles to use spin locks for resources that will be held for long periods of time. For example, a spin lock 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 sleep-type lock finds that the lock is held, it is put to sleep so that other threads can run until the lock becomes available.

The time to acquire a lock can be variable—for example, a lock needed to search and remove an item from a list. The list usually will be short, for which a spin lock would be appropriate, but will occasionally grow long. Here a hybrid lock is used; the lock spins for a while, but if unsuccessful after a specified number of attempts, it reverts to a sleep-type lock. Most architectures require 100 to 200 instructions to put a thread to sleep and then awaken it again. A spin lock that can be acquired in less than this time is going to be more efficient than a sleep lock. The hybrid lock is usually set to try for about half this time before reverting to a sleep lock.

Spin locks 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. Thus, spin locks are always converted to sleep locks when running on a uniprocessor.

The highest-level locking prevents threads from deadlocking when locking multiple resources. Suppose that two threads, A and B, require exclusive access to two resources, R_1 and R_2, to do some operation as shown in Figure 4.4. If thread A acquires R_1 and thread B acquires R_2, then a deadlock occurs when thread A tries to acquire R_2 and thread B tries to acquire R_1. To avoid deadlock, FreeBSD 5.2 maintains a partial ordering on all the locks. The two partial-ordering rules are as follows:

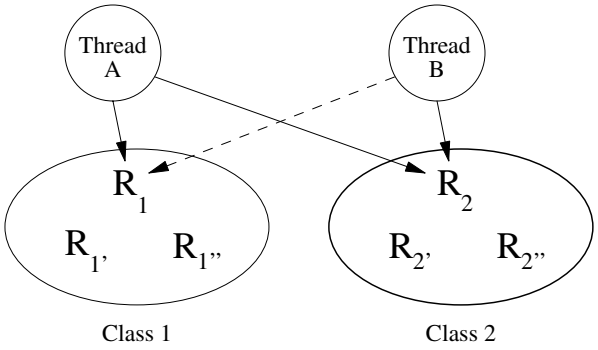


Figure 4.4 Partial ordering of resources.

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.

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 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 question. The witness module also verifies that no spin locks 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 locks. Finally, it displays a list of locks that have not yet been acquired.

Mutex Synchronization

Mutexes are the primary method of thread synchronization. The major design considerations for mutexes are

- Acquiring and releasing uncontested mutexes should be as cheap as possible.
- Mutexes must have the information and storage space to support priority propagation.
- A thread must be able to recursively acquire a mutex if the mutex is initialized to support recursion.

There are currently two flavors of mutexes: those that sleep and those that do not. By default, mutexes will sleep when they are already held. As a machine-dependent optimization they may spin for some amount of time before sleeping. Most kernel code uses the default lock type; the default lock type allows the thread to be disconnected from the CPU if it cannot get the lock. The machine-dependent implementation may treat the lock as a short-term spin lock under some circumstances. However, it is always safe to use these forms of locks in an interrupt thread without fear of deadlock against an interrupted thread on the same CPU. If a thread interrupted the thread that held a mutex and then tried to acquire the mutex, it would be put to sleep until the thread holding the mutex ran to completion and released the mutex.

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. This 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 service routine from blocking against itself all interrupts are blocked on the local processor while holding a spin lock. Thus, the interrupt can run only on another processor during the period that the mutex is held.

Spin locks are specialized locks that are intended to be held for short periods of time; their primary purpose is to protect portions of the code that implement default (i.e., sleep) locks. These mutexes should only be used to protect data shared with any devices that require nonpreemptive interrupts and low-level scheduling code. On most architectures both acquiring and releasing an uncontested spin mutex is more expensive than the same operation on a nonspin mutex. It is permissible to hold multiple spin mutexes. Here, it is a required that they be released in the opposite order to that in which they were acquired. It is not permissible to go to sleep while holding a spin mutex.

The *mtx_init*() function must be used to initialize a mutex before it can be

passed to *mtx_lock*(). 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. Interrupts are disabled on the CPU holding the mutex during the spin and remain disabled following the acquisition of the lock.

It is possible for the same thread to recursively acquire a mutex 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_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; interrupt state from before the lock was acquired is restored.

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.

The giant lock that protects subsystems in FreeBSD 5.2 that have not yet been converted to operate on a multiprocessor must be acquired before acquiring other mutexes. Put another way: It is impossible to acquire giant nonrecursively while holding another mutex. It is possible to acquire other mutexes while holding giant, and it is possible to acquire giant recursively while holding other mutexes.

Sleeping while holding a mutex (except for giant) is almost never safe and should be avoided. There are numerous assertions that will fail if this is attempted.

Lock-Manager Locks

Interprocess synchronization to a resource typically is implemented by associating it with a *lock* structure. The kernel has a lock manager that manipulates a lock. The operations provided by the lock manager are

- Request shared: Get one of many possible shared locks. If a thread holding an exclusive lock requests a shared lock, the exclusive lock will be downgraded to a shared lock.
- Request exclusive: Stop further shared locks when they are cleared, grant a pending upgrade (see following) if it exists, and then grant an exclusive lock. Only one exclusive lock may exist at a time, except that a thread holding an exclusive lock may get additional exclusive locks if it explicitly sets the *canrecurse* flag in the lock request or if the *canrecurse* flag was set when the lock was initialized.
- 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.
- 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 gotten 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, they will all be downgraded to shared locks.
- Request release: Release one instance of a lock.
- 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 the *lockinit*() function. Parameters to the *lockinit*() function include the following:

- A top-half kernel priority at which the thread should run once it has acquired the lock
- Flags such as *canrecurse* that allows 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

Other Synchronization

In addition to the general lock manager locks and mutexes, there are three other types of locking available for use in the kernel. These other types of locks are less heavily used, but they are included here for completeness.

Shared/exclusive locks are used to protect data that are read far more often than they are written. Mutexes are inherently more efficient than shared/exclusive locks. However, shared/exclusive locks are faster than lock-manager locks because they lack many of the lock-manager lock features such as the ability to be upgraded or downgraded.

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*().

Counting semaphores provide a mechanism for synchronizing access to a pool of resources. Unlike mutexes, semaphores do not have the concept of an owner, so they can also be useful in situations where one thread needs to acquire a resource and another thread needs to release it. Each semaphore has an integer value associated with it. Posting (incrementing) always succeeds, but waiting (decrementing) can only successfully complete if the resulting value of the semaphore is greater than or equal to zero. Semaphores are not used where mutexes and condition variables will suffice. Semaphores are a more complex synchronization mechanism than mutexes and condition variables, and are not as efficient.

4.4 Thread Scheduling

Traditionally, the FreeBSD scheduler had an ill-defined set of hooks spread through the kernel. In FreeBSD 5.0, these hooks were regularized and a well-defined API was created so that different schedulers could be developed. Since FreeBSD 5.0, the kernel has had two schedulers available:

- The historic 4.4BSD scheduler found in the file **/sys/kern/sched_4bsd.c**. This scheduler is described at the beginning of this section.
- The new 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 file name is removed, the rationale for its name becomes apparent. This scheduler is described at the end of this section.

Because a busy system makes thousands 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. By default, kernels up through FreeBSD 5.1 use the 4.4BSD scheduler. Beginning with FreeBSD 5.2, the ULE scheduler is used by default.

The 4.4BSD Scheduler

All threads that are runnable are assigned a scheduling priority that determines in which *run queue* they are placed. In selecting a new thread to run, the system scans the run queues 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. 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 have less overhead from doing context switches and processor caches will be flushed less often. The time quantum used by FreeBSD is 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 has remained unchanged over the past 20 years. Although the time quantum was originally selected on centralized timesharing systems with many users, it is still correct for decentralized workstations today. While workstation users expect a response time faster than that anticipated by the timesharing users of 20 years ago, the shorter run queues on the typical workstation makes a shorter quantum unnecessary.

Time-Share Thread Scheduling

The FreeBSD time-share-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.

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

Calculations of Thread Priority

A thread's scheduling priority is determined directly by two values associated with the thread structure: *kg_estcpu* and *kg_nice*. The value of *kg_estcpu* provides an estimate of the recent CPU utilization of the thread. The value of *kg_nice* is a user-settable weighting factor that ranges numerically between −20 and 20. The normal value for *kg_nice* is 0. Negative values increase a thread's priority, whereas positive values decrease its priority.

A thread's user-mode scheduling priority is calculated after every four clock ticks (typically 40 milliseconds) that it has been found running by this equation:

$$kg\_user\_pri = PRI\_MIN\_TIMESHARE + \left\lceil \frac{kg\_estcpu}{4} \right\rceil + 2 \times kg\_nice. \qquad \text{(Eq. 4.1)}$$

Values less than PRI_MIN_TIMESHARE (160) are set to PRI_MIN_TIMESHARE (see Table 4.2); values greater than PRI_MAX_TIMESHARE (223) are set to PRI_MAX_TIMESHARE. This calculation causes the priority to decrease linearly based on recent CPU utilization. The user-controllable *kg_nice* parameter acts as a limited weighting factor. Negative values retard the effect of heavy CPU utilization by offsetting the additive term containing *kg_estcpu*. Otherwise, if we ignore the second term, *kg_nice* simply shifts the priority by a constant factor.

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

$$kg\_estcpu = \frac{(2 \times load)}{(2 \times load + 1)} kg\_estcpu + kg\_nice, \qquad \text{(Eq. 4.2)}$$

where the *load* is a sampled average of the sum of the lengths of the run queue and of the short-term sleep queue over the previous 1-minute interval of system operation.

To understand the effect of the decay filter, we can consider the case where a single compute-bound thread monopolizes the CPU. The thread's CPU utilization

will accumulate clock ticks at a rate dependent on the clock frequency. The load average will be effectively 1, resulting in a decay of

$$kg\_estcpu = 0.66 \times kg\_estcpu + kg\_nice.$$

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

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

Thus, after five decay calculations, only 13 percent of T_0 remains present in the current CPU utilization value for the thread. Since the decay filter is applied once per second, we can also say that about 90 percent of the CPU utilization is forgotten after 5 seconds.

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

$$kg\_estcpu = \left[\frac{(2 \times load)}{(2 \times load + 1)} \right]^{kg\_slptime} \times kg\_estcpu, \qquad \text{(Eq. 4.3)}$$

and then recalculates the scheduling priority using Eq. 4.1. This analysis ignores the influence of *kg_nice*; also, the *load* used is the current load average rather than the load average at the time that the thread blocked.

Thread-Priority Routines

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

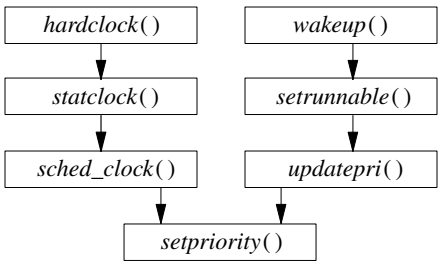
The CPU usage estimates are updated in the system clock-processing module, *hardclock*(), which executes 100 times per second. Each time that a thread accumulates four ticks in its CPU usage estimate, *kg_estcpu*, the system recalculates the priority of the thread. This recalculation uses Eq. 4.1 and is done by the *resetpriority*() routine. The decision to recalculate after four ticks is related to the management of the run queues described in the next section. In addition to issuing the call from *hardclock*(), each time *setrunnable*() places a thread on a run queue, it also calls *resetpriority*() to recompute the thread's scheduling priority. This call from *wakeup*() to *setrunnable*() operates on a thread other than the currently running thread. So *setrunnable*() invokes *updatepri*() to recalculate the CPU usage estimate according to Eq. 4.3 before calling *resetpriority*(). The relationship of these functions is shown in Figure 4.5.

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 160 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 128 to 159 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 4.6 (on page 104) 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

Figure 4.5 Procedural interface to priority calculation.



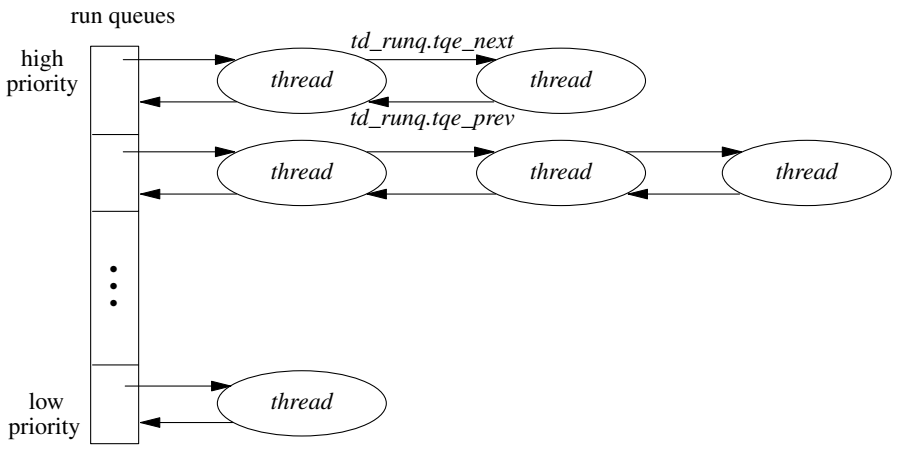


Figure 4.6 Queueing structure for runnable threads.

in identifying the nonempty run queues. Two routines, *runq_add*() and *runq_remove* (), are used to place a thread at the tail of a run queue, and to take a thread off the head of a run queue. The heart of the scheduling algorithm is the *runq_choose*() routine. The *runq_choose*() routine is responsible for selecting a new thread to run; it operates as follows:

1. Ensure that our caller acquired the *sched_lock*.
2. Locate 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 select the *idle loop* thread.
3. Given a nonempty run queue, remove the first thread on the queue.
4. If this run queue is now empty as a result of removing the thread, reset the appropriate bit in *rq_status*.
5. Return 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 by only 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 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 *swi_sched*(), *resetpriority*(), *setrunnable*(), *wakeup*(), *roundrobin*(), and *schedcpu*() routines.

With the advent of multiprocessor support FreeBSD can preempt threads executing in kernel mode. However, such preemption is generally not done, so the worst-case real-time response to events 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, FreeBSD is decidedly not a real-time system. Attempts to retrofit BSD with real-time thread scheduling have addressed this problem in different ways [Ferrin & Langridge, 1980; Sanderson et al., 1986].

The ULE Scheduler

The ULE scheduler was developed as part of the overhaul of FreeBSD to support SMP. A new scheduler was undertaken for several reasons:

- To address the need for processor affinity in SMP systems
- To provide better support for *symmetric multithreading* (SMT)—processors with multiple, on chip, CPU cores
- To improve the performance of the scheduling algorithm so that it is no longer dependent on the number of threads in the system

The goal of a multiprocessor 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 memory cache of the CPU. Migrating a thread has a cost. When a thread is moved from one processor to another, its in-cache working set is lost and must be removed from the processor on which it was running and then loaded into the new CPU to which it has been migrated. The performance of an SMP 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.

Many microprocessors now provide support for symmetric multithreading where the processor is built out of multiple CPU cores, each of which can execute a thread. The CPU cores in an SMT processor share all the processor's resources, such as memory caches and access to main memory, so they are more tightly synchronized than the processors in an SMP system. 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 cores is the scheduling algorithm. The SMT case is a slightly different version of the processor affinity problem presented by an SMP system. Each CPU core can be seen as a processor with its own set of threads. In an SMP system composed of CPUs that support SMT, the scheduler treats each core on a processor as a less powerful resource but one to which it is cheaper to migrate threads.

The original FreeBSD scheduler maintains a global list of threads that it traverses once per second to recalculate their priorities. The use of a single list for all 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. A design goal of the ULE scheduler was to avoid the need to consider all the runnable threads in the system to make a scheduling decision.

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

One queue is the *idle queue*, where all idle threads are stored. The other two queues are designated *current* and *next*. Threads are picked to run, in priority order, from the *current* queue until it is empty, at which point the *current* and *next* queues are swapped and scheduling is started again. Threads in the *idle* queue are run only when the other two queues are empty. Real-time and interrupt threads are always inserted into the *current* queue so that they will have the least possible scheduling latency. Interactive threads are also inserted into the *current* queue to keep the interactive response of the system acceptable. 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 4.4 is used:

$$interactivity\ score = \frac{scaling\ \ factor}{\frac{sleep}{run}} \qquad \text{(Eq. 4.4)}$$

When a thread's run time exceeds its sleep time, Eq. 4.5 is used instead.

$$interactivity\ score = \frac{scaling\ \ factor}{\frac{run}{sleep}} + scaling\ \ factor \qquad \text{(Eq. 4.5)}$$

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 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.

Noninteractive threads are put into the *next* queue and are scheduled to run when the queues are switched. Switching the queues guarantees that a thread gets to run at least once every two queue switches regardless of priority, which ensures fair sharing of the processor.

There are two mechanisms used to migrate threads among multiple processors. When a CPU has no work to do in any of its queues, it marks a bit in a bitmask shared by all processors that says that it is idle. 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 processor in the system is idle. If an idle processor is found, then the thread is migrated to the idle processor 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 second 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. Twice per second the *sched_balance*() routine picks the most-loaded and least-loaded processors in the system and equalizes their run queues. The balancing is done only between two processors because it was thought that two processor systems would be the most common and to prevent the balancing algorithm from being too complex and adversely affecting the performance of the scheduler. 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. 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.

Handling the SMT case is a derivative form of load balancing among full-fledged CPUs and is handled by *processor groups*. Each CPU core in an SMT processor is given its own *kseq* structure, and these structures are grouped under a *kseq group* structure. An example of a single processor with two cores is shown in Figure 4.7 (on page 108). In an SMP system with multiple SMT capable processors there would be one processor group per CPU. When the scheduler is deciding to which processor or core to migrate a thread, it will try to pick a core on the

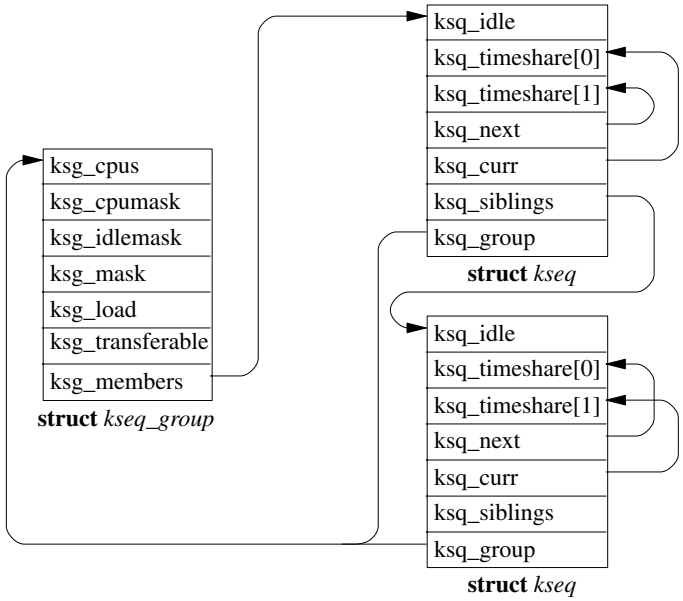


Figure 4.7 Processor with two cores.

same processor before picking one on another processor because that is the lowest-cost migration path.

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 5.6.

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 5. 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 *kg_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 explicitly set information includes

- 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 pointer to the process's statistics structure, allocated in its user structure
- A pointer to the process's signal-actions structure, allocated in its user structure
- 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 user space 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 to back out to user space, freeing resources as they go. When the thread attempts to return to user space, it will instead hit *thread_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 processes 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 would 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

UNIX defines a set of *signals* for software and hardware conditions that may arise during the normal execution of a program; these signals are listed in Table 4.4 (on page 112). Signals may be delivered to a process through application-specified *signal handlers* or may result in *default* actions, such as process termination, carried out by the system. FreeBSD signals are designed to be software equivalents of hardware interrupts or traps.

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

Table 4.4 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

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 4.4) 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:

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

History of 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. Thus, 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 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 4.5.

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. But 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*()

Table 4.5 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

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. So 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.

The *cursig*() routine calculates the next signal, if any, that should be delivered to a thread. It determines the next signal 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. 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 checks to see whether a signal is pending delivery. 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 a *continue signal*, SIGCONT, any pending signals that would normally cause the process to stop, such as SIGTTOU, are removed.
3. Next, *psignal*() checks whether the signal is being masked. If the thread is currently masking delivery of the signal, *psignal*()'s work is complete and it may return.
4. If, however, the signal is not being masked, *psignal*() must either do 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 thread state 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.

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

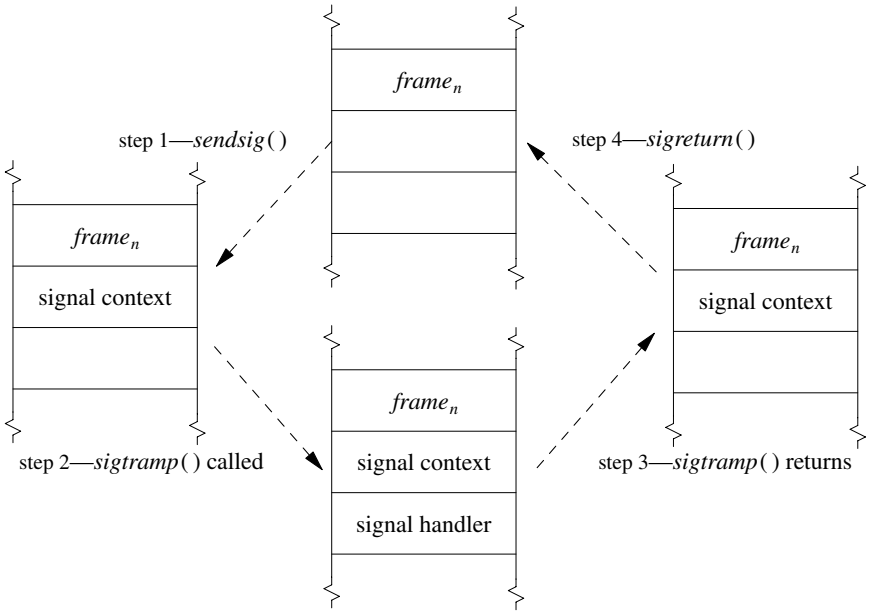


Figure 4.8 Delivery of a signal to a process. Step 1: The kernel places a signal context on the user's stack. Step 2: The kernel places a signal-handler frame on the user's stack and arranges to start running the user process in the *sigtramp*( ) code. When the *sigtramp*( ) routine starts running, it 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.

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 4.8 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 4.8) 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.

4.8 Process Groups and Sessions

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 name space 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 a 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 its own PID or 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 yet if a child exits. As long as a process-group member exists, even as a zombie process, additional processes can join the process group.

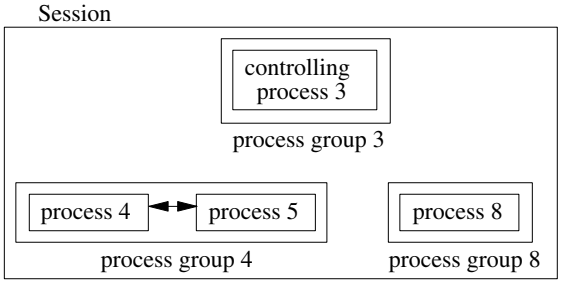
There are additional restrictions on the *setpgid* system call. A process may join process groups only within its current session (discussed in the next section), and it cannot have done an *exec* system call. The latter restriction is intended to avoid unexpected behavior if a process is moved into 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 together 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

Figure 4.9 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, processes 3, 4, or 8 could not start its own session, but process 5 would be allowed to do so.



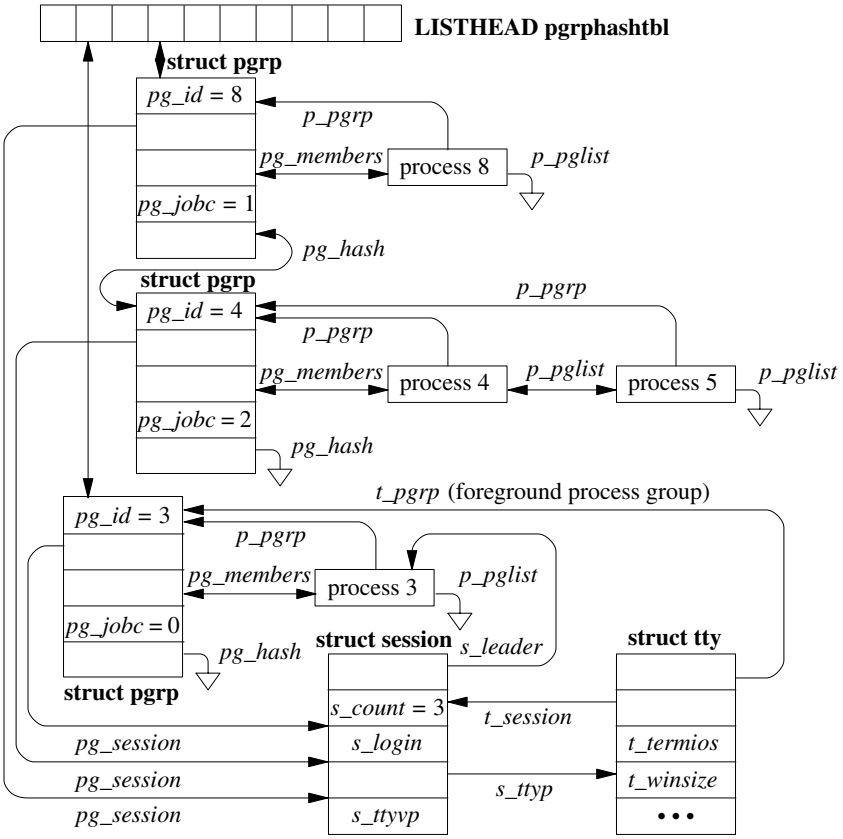


Figure 4.10 Process-group organization.

device can be the controlling terminal for only one session at a time. The terminal I/O system (described in Chapter 10) 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 4.9.

The data structures used to support sessions and process groups in FreeBSD are shown in Figure 4.10. This figure parallels the process layout shown in Figure 4.9. 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 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 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 (Chapter 10) and the interprocess-communication facilities (Chapter 11). 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, and, if the TOSTOP option is set for the terminal, if they attempt 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 Chapter 10). All other process groups within the session are in the background. In Figure 4.10, 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 4.10 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, attempts to read or write to the terminal produce an error.

4.9 Jails

The FreeBSD access control mechanism is designed for an environment with two types of users: those with and those without administrative privilege. It is often desirable to delegate some but not all administrative functions to untrusted or less trusted parties and simultaneously impose systemwide mandatory policies on process interaction and sharing. Historically, attempting to create such an environment has been both difficult and costly. The primary mechanism for partial delegation of administrative authority is to write a set-user-identifier program that carefully controls which of the administrative privileges may be used. These set-user-identifier programs are complex to write, difficult to maintain, limited in their flexibility, and are prone to bugs that allow undesired administrative privilege to be gained.

Many operating systems attempt to address these limitations by providing fine-grained access controls for system resources [P1003.1e, 1998]. These efforts

vary in degrees of success, but almost all suffer from at least three serious limitations:

1. Increasing the granularity of security controls increases the complexity of the administration process, in turn increasing both the opportunity for incorrect configuration, as well as the demand on administrator time and resources. Often the increased complexity results in significant frustration for the administrator, which may result in two disastrous types of policy: running with security features disabled and running with the default configuration on the assumption that it will be secure.
2. Usefully segregating capabilities and assigning them to running code and users is difficult. Many privileged operations in FreeBSD seem independent but are interrelated. The handing out of one privilege may be transitive to many others. For example, the ability to mount filesystems allows new set-user-identifier programs to be made available that in turn may yield other unintended security capabilities.
3. Introducing new security features often involves introducing new security management interfaces. When fine-grained capabilities are introduced to replace the set-user-identifier mechanism in FreeBSD, applications that previously did an appropriateness check to see if they were running with superuser privilege before executing must now be changed to know that they need not run with superuser privilege. For applications running with privilege and executing other programs, there is now a new set of privileges that must be voluntarily given up before executing another program. These changes can introduce significant incompatibility for existing applications and make life more difficult for application developers who may not be aware of differing security semantics on different systems.

This abstract risk becomes more clear when applied to a practical, real-world example: many Web service providers use FreeBSD to host customer Web sites. These providers must protect the integrity and confidentiality of their own files and services from their customers. They must also protect the files and services of one customer from (accidental or intentional) access by any other customer. A provider would like to supply substantial autonomy to customers, allowing them to install and maintain their own software and to manage their own services such as Web servers and other content-related daemon programs.

This problem space points strongly in the direction of a partitioning solution. By putting each customer in a separate partition, customers are isolated from accidental or intentional modification of data or disclosure of process information from customers in other partitions. Delegation of management functions within the system must be possible without violating the integrity and privacy protection between partitions.

FreeBSD-style access control makes it notoriously difficult to compartmentalize functionality. While mechanisms such as *chroot* provide a modest level of

compartmentalization, this mechanism has serious shortcomings, both in the scope of its functionality and effectiveness at what it provides. The *chroot* system call was first added to provide an alternate build environment for the system. It was later adapted to isolate anonymous **ftp** access to the system.

The original intent of *chroot* was not to ensure security. Even when used to provide security for anonymous **ftp**, the set of operations allowed by **ftp** was carefully controlled to prevent those that allowed escape from the *chroot*'ed environment.

Three classes of escape from the confines of a *chroot*-created filesystem were identified over the years:

1. Recursive *chroot* escapes
2. Escapes using **..**
3. Escapes using *fchdir*

All these escapes exploited the lack of enforcement of the new root directory.

Two changes to *chroot* were made to detect and thwart these escapes. To prevent the first two escapes, the directory of the first level of *chroot* experienced by a process is recorded. Any attempts to traverse backward across this directory are refused. The third escape using *fchdir* is prevented by having the *chroot* system call fail if the process has any file descriptors open referencing directories.

Even with stronger semantics, the *chroot* system call is insufficient to provide complete partitioning. Its compartmentalization does not extend to the process or networking spaces. Therefore, both observation of and interference with processes outside their compartment is possible. To provide a secure virtual machine environment, FreeBSD added a new ''jail'' facility built on top of *chroot*. Processes in a jail are provided full access to the files that they may manipulate, processes they may influence, and network services they may use. They are denied access to and visibility of files, processes, and network services outside their jail [Kamp & Watson, 2000].

Unlike other fine-grained security solutions, a jail does not substantially increase the policy management requirements for the system administrator. Each jail is a virtual FreeBSD environment that permits local policy to be independently managed. The environment within a jail has the same properties as the host system. Thus, a jail environment is familiar to the administrator and compatible with applications [Hope, 2002].

Jail Semantics

Two important goals of the jail implementation are to:

1. Retain the semantics of the existing discretionary access-control mechanisms
2. Allow each jail to have its own superuser administrator whose activities are limited to the processes, files, and network associated with its jail

The first goal retains compatibility with most applications. The second goal permits the administrator of a FreeBSD machine to partition the host into separate jails and provide access to the superuser account in each of these jails without losing control of the host environment.

A process in a partition is referred to as being ''in jail.'' When FreeBSD first boots, no processes will be jailed. Jails are created when a privileged process calls the *jail* system call with arguments of the filesystem into which it should *chroot* and the IP address and hostname to be associated with the jail. The process that creates the jail will be the first and only process placed in the jail. Any future descendants of the jailed process will be in its jail. A process may never leave a jail that it created or in which it was created. A process may be in only one jail. The only way for a new process to enter the jail is by inheriting access to the jail from another process already in that jail.

Each jail is bound to a single IP address. Processes within the jail may not make use of any other IP address for outgoing or incoming connections. A jail has the ability to restrict the set of network services that it chooses to offer at its address. An application request to bind all IP addresses are redirected to the individual address associated of the jail in which the requesting process is running.

A jail takes advantage of the existing *chroot* behavior to limit access to the filesystem name space for jailed processes. When a jail is created, it is bound to a particular filesystem root. Processes are unable to manipulate files that they cannot address. Thus, the integrity and confidentiality of files outside the jail filesystem root are protected.

Processes within the jail will find that they are unable to interact or even verify the existence of processes outside the jail. Processes within the jail are prevented from delivering signals to processes outside the jail, connecting to processes outside the jail with debuggers, or even seeing processes outside the jail with the usual system monitoring mechanisms. Jails do not prevent, nor are they intended to prevent, the use of covert channels or communications mechanisms via accepted interfaces. For example, two processes in different jails may communicate via sockets over the network. Jails do not attempt to provide scheduling services based on the partition.

Jailed processes are subject to the normal restrictions present for any processes including resource limits and limits placed by the network code, including firewall rules. By specifying firewall rules for the IP address bound to a jail, it is possible to place connectivity and bandwidth limitations on that jail, restricting the services that it may consume or offer.

The jail environment is a subset of the host environment. The jail filesystem appears as part of the host filesystem and may be directly modified by processes in the host environment. Processes within the jail appear in the process listing of the host and may be signalled or debugged.

Processes running without superuser privileges will notice few differences between a jailed environment or an unjailed environment. Standard system services such remote login and mail servers behave normally as do most third-party applications, including the popular Apache Web server.

Processes running with superuser privileges will find that many restrictions apply to the privileged calls they may make. Most of the limitations are designed to restrict activities that would affect resources outside the jail. These restrictions include prohibitions against the following:

- Modifying the running kernel by direct access or loading kernel modules
- Mounting and unmounting filesystems
- Creating device nodes
- Modifying kernel run-time parameters such as most sysctl settings
- Changing security-level flags
- Modifying any of the network configuration, interfaces, addresses, and routing-table entries
- Accessing raw, divert, or routing sockets. These restrictions prevent access to facilities that allow spoofing of IP numbers or the generation of disruptive traffic.
- Accessing network resources not associated with the jail. Specifically, an attempt to bind a reserved port number on all available addresses will result in binding only the address associated with the jail.
- Administrative actions that would affect the host system such as rebooting

Other privileged activities are permitted as long as they are limited to the scope of the jail:

- Signalling any process within the jail is permitted.
- Deleting or changing the ownership and mode of any file within the jail is permitted, as long as the file flags permit the requested change.
- The superuser may read a file owned by any UID, as long as it is accessible through the jail filesystem name space.
- Binding reserved TCP and UDP port numbers on the jail's IP address is permitted.

These restrictions on superuser access limit the scope of processes running with superuser privileges, enabling most applications to run unhindered but preventing calls that might allow an application to reach beyond the jail and influence other processes or systemwide configuration.

Jail Implementation

The implementation of the *jail* system call is straightforward. A *prison* data structure is allocated and populated with the arguments provided. The prison structure is linked to the process structure of the calling process. The prison structure's reference count is set to one, and the *chroot* system call is called to set the jail's root. The prison structure may not be modified once it is created.

Hooks in the code implementing process creation and destruction maintain the reference count on the prison structure and free it when the last reference is released. Any new processes created by a process in a jail will inherit a reference to the prison structure, which puts the new process in the same jail.

Some changes were needed to restrict process visibility and interaction. The kernel interfaces that report running processes were modified to report only the processes in the same jail as the process requesting the process information. Determining whether one process may send a signal to another is based on UID and GID values of the sending and receiving processes. With jails, the kernel adds the requirement that if the sending process is jailed, then the receiving process must be in the same jail.

Several changes were added to the networking implementation:

- Restricting TCP and UDP access to just one IP number was done almost entirely in the code that manages protocol control blocks (see Section 13.1). When a jailed process binds to a socket, the IP number provided by the process will not be used; instead, the preconfigured IP number of the jail is used.
- The loop-back interface, which has the magic IP number 127.0.0.1, is used by processes to contact servers on the local machine. When a process running in a jail connects to the 127.0.0.1 address, the kernel must intercept and redirect the connection request to the IP address associated with the jail.
- The interfaces through which the network configuration and connection state may be queried were modified to report only information relevant to the configured IP number of a jailed process.

Device drivers for shared devices such as the pseudo-terminal driver (see Section 10.1) needed to be changed to enforce that a particular virtual terminal cannot be accessed from more than one jail at the same time.

The simplest but most tedious change was to audit the entire kernel for places that allowed the superuser extra privilege. Only about 35 of the 300 checks in FreeBSD 5.2 were opened to jailed processes running with superuser privileges. Since the default is that jailed superusers do not receive privilege, new code or drivers are automatically jail-aware: They will refuse jailed superusers privilege.

Jail Limitations

As it stands, the jail code provides a strict subset of system resources to the jail environment, based on access to processes, files, network resources, and privileged services. Making the jail environment appear to be a fully functional FreeBSD system allows maximum application support and the ability to offer a wide range of services within a jail environment. However, there are limitations in the current implementation. Removing these limitations will enhance the ability to offer services in a jail environment. Three areas that deserve greater attention are the set of available network resources, management of scheduling resources, and support for orderly jail shutdown.

Currently, only a single IP version 4 address may be allocated to each jail, and all communication from the jail is limited to that IP address. It would be desirable to support multiple addresses or possibly different address families for each jail. Access to raw sockets is currently prohibited, as the current implementation of raw sockets allows access to raw IP packets associated with all interfaces. Limiting the scope of the raw socket would allow its safe use within a jail, thus allowing the use of **ping** and other network debugging and evaluation tools.

Another area of great interest to the current users of the jail code is the ability to limit the effect of one jail on the CPU resources available for other jails. Specifically, they require that the system have ways to allocate scheduling resources among the groups of processes in each of the jails. Work in the area of lottery scheduling might be leveraged to allow some degree of partitioning between jail environments [Petrou & Milford, 1997].

Management of jail environments is currently somewhat ad hoc. Creating and starting jails is a well-documented procedure, but jail shutdown requires the identification and killing of all the processes running within the jail. One approach to cleaning up this interface would be to assign a unique jail-identifier at jail creation time. A new *jailkill* system call would permit the direction of signals to specific jail-identifiers, allowing for the effective termination of all processes in the jail. FreeBSD makes use of an **init** process to bring the system up during the boot process and to assist in shutdown (see Section 14.6). A similarly operating process, **jailinit**, running in each jail would present a central location for delivering management requests to its jail from the host environment or from within the jail. The **jailinit** process would coordinate the clean shutdown of the jail before resorting to terminating processes, in the same style as the host environment shutting down before killing all processes and halting the kernel.

4.10 Process Debugging

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

- Attach to an existing process to begin debugging it
- Read and write address space and registers
- Intercept signals posted to the process
- Single step and continue the execution of the process
- Terminate the execution of the process

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

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** Continue execution of the target process and remove it from control by the debug process (that need not be the sending process).
- **run** Continue running the target process until a signal is delivered, a breakpoint is hit, or the target process exits.
- **step** Single step the target process, with no signal delivery.

wait Wait 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 Set 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 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*.

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 time-share 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 Figure 4.2 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 name space, 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.

Ferrin & Langridge, 1980.

T. E. Ferrin & R. Langridge, ''Interactive Computer Graphics with the UNIX Time-Sharing System,'' *Computer Graphics*, vol. 13, pp. 320–331, 1980.

Hope, 2002.

P. Hope, ''Using Jails in FreeBSD for Fun and Profit,'' *;login: The USENIX Association Newsletter*, vol. 27, no. 3, pp. 48–55, available from http://www.usenix.org/publications/login/2002-06/pdfs/hope.pdf, USENIX Association, Berkeley, CA, June 2002.

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.

Kamp & Watson, 2000.

P. Kamp & R. Watson, ''Jails: Confining the Omnipotent Root,'' *Proceedings of the Second International System Administration and Networking Conference (SANE)*, available from http://docs.freebsd.org/44doc/papers/jail/, May 2000.

P1003.1e, 1998.

P1003.1e, *Unpublished Draft Standard for Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface—Amendment: Protection, Audit and Control Interfaces [C Language] IEEE Standard 1003.1e Draft 17 Editor Casey Schaufler,* Institute of Electrical and Electronic Engineers, Piscataway, NJ, 1998.

Petrou & Milford, 1997.

D. Petrou & J. Milford, *Proportional-Share Scheduling: Implementation and Evaluation in a Widely-Deployed Operating System,* available from http://www.cs.cmu.edu/~dpetrou/papers/freebsd_lottery_writeup98.ps and http://www.cs.cmu.edu/~dpetrou/code/freebsd_lottery_code.tar.gz, 1997.

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 BSDCon 2003*, September 2003.

Sanderson et al., 1986.

T. Sanderson, S. Ho, N. Heijden, E. Jabs, & J. L. Green, ''Near-Realtime Data Transmission During the ICE-Comet Giacobini-Zinner Encounter,'' *ESA Bulletin*, vol. 45, no. 21, 1986.

Schimmel, 1994.

C. Schimmel, *UNIX Systems for Modern Architectures, Symmetric Multiprocessing, and Caching for Kernel Programmers,* Addison-Wesley, Reading, MA, 1994.