level calls return the previous priority level. When the critical section is done, the priority is returned to its previous level using *splx*( ). For example, when a process needs to manipulate a terminal's data queue, the code that accesses the queue is written in the following style:

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

Processes must take care to avoid deadlocks when locking multiple resources. Suppose that two processes, A and B, require exclusive access to two resources, R_1 and R_2, to do some operation. If process A acquires R_1 and process B acquires R_2, then a deadlock occurs when process A tries to acquire R_2 and process B tries to acquire R_1. Since a 4.4BSD process executing in kernel mode is never preempted by another process, locking of multiple resources is simple, although it must be done carefully. If a process knows that multiple resources are required to do an operation, then it can safely lock one or more of those resources in any order, as long as it never relinquishes control of the CPU. If, however, a process cannot acquire all the resources that it needs, then it must release any resources that it holds before calling *sleep*( ) to wait for the currently inaccessible resource to become available.

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

Raising the processor priority level to guard against interrupt activity works for a uniprocessor architecture, but not for a shared-memory multiprocessor machine. Similarly, much of the 4.4BSD kernel implicitly assumes that kernel processing will never be done concurrently. Numerous vendors—such as Sequent, OSF/1, AT&T, and Sun Microsystems—have redesigned the synchronization schemes and have eliminated the uniprocessor assumptions implicit in the standard UNIX kernel, so that UNIX will run on tightly coupled multiprocessor architectures [Schimmel, 1994].

4.4 Process Scheduling

4.4BSD uses a process-scheduling algorithm based on *multilevel feedback queues*. All processes that are runnable are assigned a scheduling priority that determines in which *run queue* they are placed. In selecting a new process to run, the system scans the run queues from highest to lowest priority and chooses the first process

on the first nonempty queue. If multiple processes reside on a queue, the system runs them *round robin*; that is, it runs them in the order that they are found on the queue, with equal amounts of time allowed. If a process blocks, it is not put back onto any run queue. If a process uses up the *time quantum* (or *time slice*) allowed it, it is placed at the end of the queue from which it came, and the process at the front of the queue is selected to run.

The shorter the time quantum, the better the interactive response. However, longer time quanta provide higher system throughput, because the system will have less overhead from doing context switches, and processor caches will be flushed less often. The time quantum used by 4.4BSD 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 15 years. Although the time quantum was originally selected on centralized timesharing systems with many users, it is still correct for decentralized workstations today. Although workstation users expect a response time faster than that anticipated by the time-sharing users of 10 years ago, the shorter run queues on the typical workstation makes a shorter quantum unnecessary.

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

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

Calculations of Process Priority

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

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

$$p\_usrpri = PUSER + \left[\frac{p\_estcpu}{4} \right] + 2 \times p\_nice. \qquad \text{(Eq. 4.1)}$$

Values less than PUSER are set to PUSER (see Table 4.2); values greater than 127 are set to 127. This calculation causes the priority to decrease linearly based on recent CPU utilization. The user-controllable *p_nice* parameter acts as a limited weighting factor. Negative values retard the effect of heavy CPU utilization by offsetting the additive term containing *p_estcpu*. Otherwise, if we ignore the second term, *p_nice* simply shifts the priority by a constant factor.

The CPU utilization, *p_estcpu*, is incremented each time that the system clock ticks and the process is found to be executing. In addition, *p_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, *p_estcpu* is adjusted according to

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

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 process monopolizes the CPU. The process's CPU utilization will accumulate clock ticks at a rate dependent on the clock frequency. The load average will be effectively 1, resulting in a decay of

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

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

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

Thus, after five decay calculations, only 13 percent of T_0 remains present in the current CPU utilization value for the process. 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.

Processes that are runnable have their priority adjusted periodically as just described. However, the system ignores processes blocked awaiting an event: These processes cannot accumulate CPU usage, so an estimate of their filtered CPU usage can be calculated in one step. This optimization can significantly reduce a system's scheduling overhead when many blocked processes are present. The system recomputes a process's priority when that process is awakened and

has been sleeping for longer than 1 second. The system maintains a value, *p_slptime*, that is an estimate of the time a process has spent blocked waiting for an event. The value of *p_slptime* is set to 0 when a process calls *sleep*(), and is incremented once per second while the process remains in an SSLEEP or SSTOP state. When the process is awakened, the system computes the value of *p_estcpu* according to

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

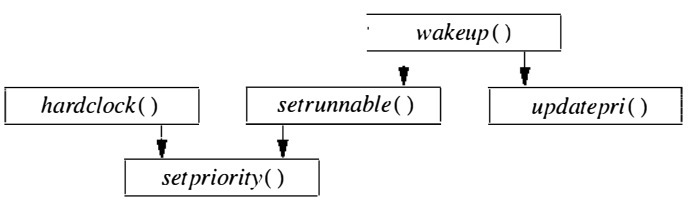
and then recalculates the scheduling priority using Eq. 4.1. This analysis ignores the influence of *p_nice*; also, the *load* used is the current load average, rather than the load average at the time that the process blocked.

Process-Priority Routines

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

The CPU usage estimates are updated in the system clock-processing module, *hardclock*(), which executes 100 times per second. Each time that a process accumulates four ticks in its CPU usage estimate, *p_estcpu*, the system recalculates the priority of the process. This recalculation uses Eq. 4.1 and is done by the *setpriority*() 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 process on a run queue, it also calls *setpriority*() to recompute the process's scheduling priority. This call from *wakeup*() to *setrunnable*() operates on a process other than the currently running process. So, *wakeup*() invokes *updatepri*() to recalculate the CPU usage estimate according to Eq. 4.3 before calling *setpriority*(). The relationship of these functions is shown in Fig. 4.4.

Figure 4.4 Procedural interface to priority calculation.

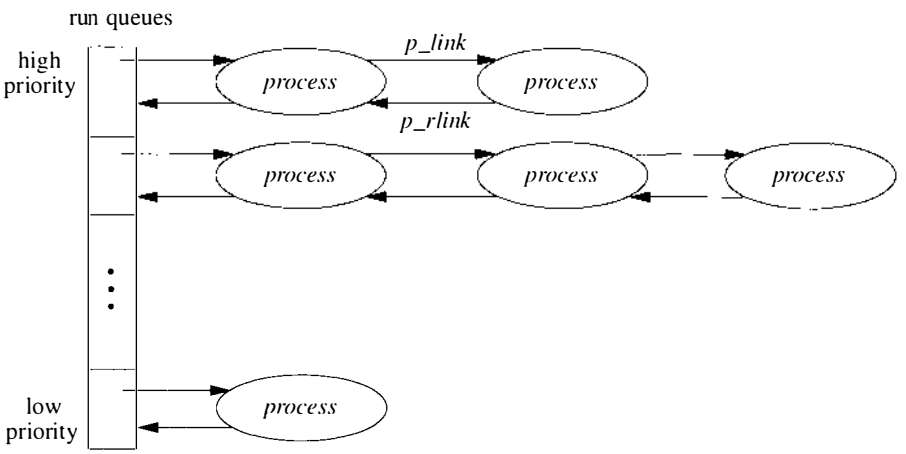


Process Run Queues and Context Switching

The scheduling-priority calculations are used to order the set of runnable processes. The scheduling priority ranges between 0 and 127, with 0 to 49 reserved for processes executing in kernel mode, and 50 to 127 reserved for processes executing in user mode. The number of queues used to hold the collection of runnable processes affects the cost of managing the queues. If only a single (ordered) queue is maintained, then selecting the next runnable process becomes simple, but other operations become expensive. Using 128 different queues can significantly increase the cost of identifying the next process to run. The system uses 32 run queues, selecting a run queue for a process by dividing the process's priority by 4. The processes on each queue are not further sorted by their priorities. The selection of 32 different queues was originally a compromise based mainly on the availability of certain VAX machine instructions that permitted the system to implement the lowest-level scheduling algorithm efficiently, using a 32-bit mask of the queues containing runnable processes. The compromise works well enough today that 32 queues are still used.

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

Figure 4.5 Queueing structure for runnable processes.



1. Block interrupts, then look for a nonempty run queue. Locate a nonempty queue by finding the location of the first nonzero bit in the *whichqs* bit vector. If *whichqs* is zero, there are no processes to run, so unblock interrupts and loop; this loop is the *idle loop*.
2. Given a nonempty run queue, remove the first process on the queue.
3. If this run queue is now empty as a result of removing the process, reset the appropriate bit in *whichqs*.
4. Clear the *curproc* pointer and the *want_resched* flag. The *curproc* pointer references the currently running process. Clear it to show that *no process is currently running*. The *want_resched* flag shows that a context switch should take place; it is described later in this section.
5. Set the new process running and unblock interrupts.

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 process-priority calculations, the only missing piece in the scheduling facility is how the system forces an involuntary context switch. Remember that voluntary context switches occur when a process calls the *sleep*() routine. *Sleep*() can be invoked by only a runnable process, so *sleep*() needs only to place the process on a sleep queue and to invoke *mi_switch*() to schedule the next process to run. The *mi_switch*() routine, however, cannot be called from code that executes at interrupt level, because it must be called within the context of the running process.

An alternative mechanism must exist. This mechanism is handled by the machine-dependent *need_resched*() routine, which generally sets a global *reschedule request* flag, named *want_resched*, and then posts an *asynchronous system trap* (*AST*) for the current process. An AST is a trap that is delivered to a process the next time that that process returns to user mode. Some architectures support ASTs directly in hardware; other systems emulate ASTs by checking the *want_resched* flag at the end of every system call, trap, and interrupt of user-mode execution. When the hardware AST trap occurs or the *want_resched* flag is set, the *mi_switch*() routine is called, instead of the current process resuming execution. Rescheduling requests are made by the *wakeup*(), *setpriority*(), *roundrobin*(), *schedcpu*(), and *setrunnable*() routines.

Because 4.4BSD does not preempt processes executing in kernel mode, 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, 4.4BSD is decidedly not a real-time system. Attempts to retrofit BSD with real-time process scheduling have addressed this problem in different ways [Ferrin & Langridge, 1980; Sanderson et al, 1986].