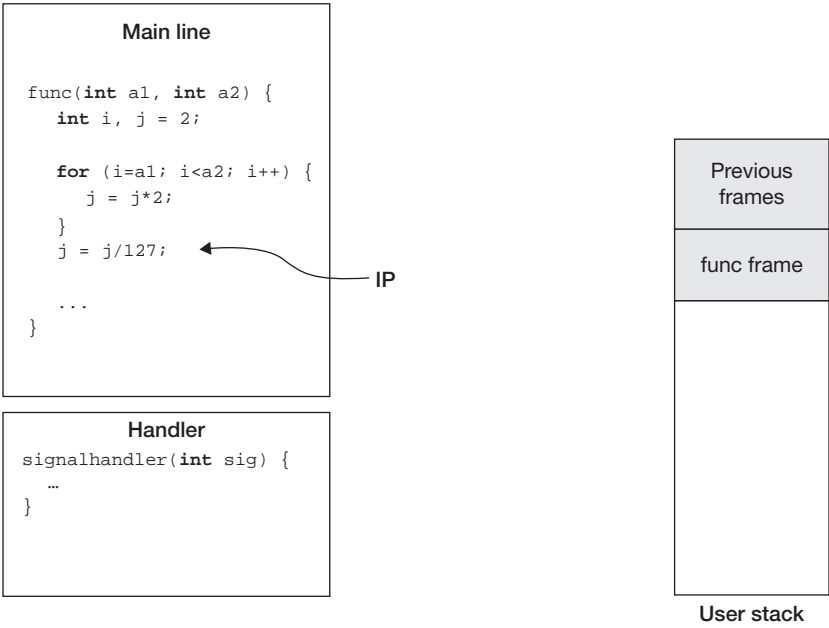


**FIGURE 5.14** On return from the signal handler, the thread's instruction pointer points to the *sigreturn* instruction.



**FIGURE 5.15** Finally, the thread executes the *sigreturn* system call, causing the kernel to restore the thread's original registers and unblock the signal.

5.3  
SCHEDULING

A common notion of what an operating system does is that it manages resources — it determines who or what is to have which resource when. Processor time is apportioned to threads. Primary memory is apportioned to processes. Disk space is apportioned to users. On some data-processing systems, I/O bandwidth is apportioned to jobs or subsystems.

Our concern in this section is the sharing of processors, a task that's usually referred to as *scheduling* (though certain aspects of memory management in some Unix systems are called

“memory scheduling”). The common use of the term *schedule* is that it’s something that’s prepared ahead of time — we have a schedule for the use of a classroom or conference room; an airline has a flight schedule. Such static schedules make sense if we know or can predict the demands in advance. But the sort of scheduling we’re primarily interested in here is dynamic: responding to demands for immediate use of processor time. At any one moment we might have a static schedule for at least the order in which threads wait for processor time, but this schedule changes in response to additional threads becoming runnable and other system events.

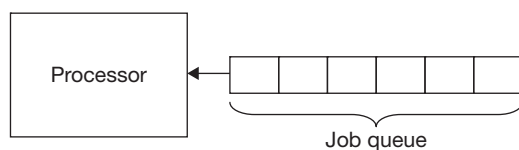
A lot goes into determining schedules. At the strategic level, we are trying to make a “good” decision based on some sort of optimization criteria. Are we trying to give good response to interactive threads? Are we trying to give deterministic (and good) response to real-time threads? Are we trying to maximize the number of jobs per hour? Are we trying to do all of the above?

At the tactical level, we need to organize our list of runnable threads so as to find the next thread to run quickly. On multiprocessor systems we need to take into account the benefits of caching: a thread runs best on a processor whose cache contains much of what that thread is using. Furthermore, we must consider the cost of synchronization and organize our data structures to minimize such costs.

### 5.3.1 STRATEGY

How the processor is shared depends upon what the system as a whole is supposed to do. Listed below are five representative types of systems along with brief descriptions of the sort of sharing desired.

- *Simple batch systems.* These probably don’t exist anymore, but they were common into the 1960s. Programs (jobs) were submitted and ran without any interaction with humans, except for possible instructions to the operator to mount tapes and disks. Only one job ran at a time. The basic model is shown in Figure 5.16: a queue of jobs waiting to be run on the processor. The responsibility of the scheduler was to decide which job should run next when the current one finished. There were two concerns: the system throughput, i.e., the number of jobs per unit time, and the average wait time, i.e., how long it took from when a job was submitted to the system until it completed.
- *Multiprogrammed batch systems.* These are identical to simple batch systems except that multiple jobs are run concurrently. Two sorts of scheduling decisions have to be made: how many and which jobs should be running, and how the processor is apportioned among the running jobs.
- *Time-sharing systems.* Here we get away from the problem of how many and which jobs should be running and think more in terms of apportioning the processor to the threads that are ready to execute. The primary concern is wait time, here called response time — the time from when a command is given to when it is completed. Short requests should be handled quickly.
- *Shared servers.* This is the modern version of the multiprogrammed batch system. A single computer is used concurrently by a number of clients, each getting its fair share. For example, a large data-processing computer might be running a number of different online systems each



**FIGURE 5.16** A simple batch system.

of which must be guaranteed a certain capacity or performance level — we might want to guarantee each at least 10% of available processing time. A web-hosting service might want to give each of its clients the same fraction of total processor time, regardless of the number of threads employed.

- *Real-time systems.* These have a range of requirements, ranging from what’s known as “soft” real-time to “hard” real-time. An example of the former is a system to play back streaming audio or video. It’s really important that most of the data be processed in a timely fashion, but it’s not a disaster if occasionally some data isn’t processed on time (or at all). An example of the latter is a system controlling a nuclear reactor. It’s not good enough for it to handle most of the data in a timely fashion; it must handle *all* the data in a timely fashion or there will be a disaster.

5.3.1.1 Simple Batch Systems

Much of the early work on scheduling dealt with the notion of jobs — work to be done, usually by a single-threaded program, whose running time was known. In this sort of situation one could conceivably come up with an optimal static schedule.

On a simple batch system (as described above) you might think we can do no better than *first-in-first-out (FIFO)* scheduling, i.e., a simple queue. However, if our sole criterion for goodness of our scheduler is the number of jobs completed per hour, this strategy could get us into trouble: if the first job takes a week (168 hours) to complete, but the following 168 jobs each take an hour, our completion statistics aren’t going to look very good, at least not until towards the end of the second week (see Figure 5.17).

From the point of view of the people who submitted the jobs, a relevant measure of “goodness” might be the average amount of time the submitters had to wait between submitting their job and its completion. If all jobs were submitted at roughly the same time, but the long one was submitted first, the average waiting time is 252 hours: the submitter of the first job waited 168 hours, the submitter of the second 169 hours, the submitter of the third 170 hours, and so forth. Summing up these wait times and dividing by the number of jobs (169) yields 252.

A better strategy might be *shortest-job-first (SJF)* (Figure 5.18): whenever we must choose which job to run next, we choose the one requiring the least running time. Thus in the example of the previous paragraph, rather than having to report 0 jobs/hour completed during the first week, we can report 1 job/hour. With both approaches the figure at the end of the second week is .503 jobs/hour. However, the average wait time is now down to 86 hours. Of course, if we continue to get more of these one-hour jobs, the one-week job might never be handled, but if our concern is solely throughput, we don’t care.

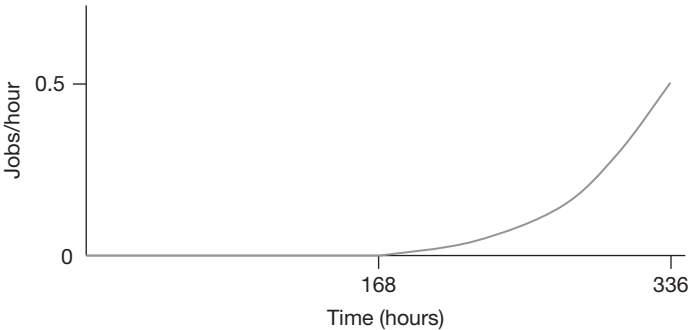


FIGURE 5.17 FIFO scheduling applied to our sample workload.

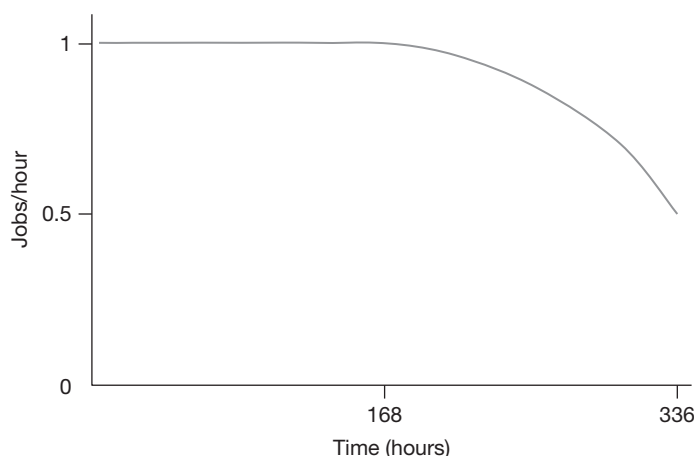


FIGURE 5.18 SJF scheduling applied to our sample workload.

### 5.3.1.2 Multiprogrammed Batch Systems

Suppose a multiprogrammed batch system runs two jobs concurrently, using two FIFO queues. In one queue are placed long-running jobs and in the other short-running jobs. So, continuing with our example, the 168-hour job is placed in the first queue and the one-hour jobs in the other. When two jobs share the processor, their execution is *time-sliced*: each runs for a certain period of time, known as the *time quantum*, then is preempted in favor of the other.

As with the simple batch systems, the throughput after two weeks is .503 jobs/hour. Computing the average wait time given the time quantum is a bit cumbersome, but consider what happens as the quantum approaches zero: each job experiences a processor that's effectively half its normal speed. Thus the 168-hour job takes two weeks to complete and each of the one-hour jobs takes two hours. The short jobs have an average wait time of 169 hours, while the overall average wait time is 169.99 hours. This is not as good as what we obtained with SJF, but it's better than FIFO and the long job makes progress even if we have a large number of short jobs.

### 5.3.1.3 Time-Sharing Systems

On time-sharing systems the primary scheduling concern is that the system appear responsive to interactive users. This means that operations that should be quick really are. Users aren't overly annoyed if something that normally takes 2 minutes to run, such as building a large system, takes 5 minutes. But if something that normally seems instantaneous, such as adding a line to a file when running an editor, starts taking more than a second, interactive response is considered poor. Thus in a time-sharing system we want a scheduling strategy that favors short operations at the possible expense of longer ones.

A simple time-sharing scheduler might be time-sliced and employ a single round-robin run queue: a running thread that completes its time slice is put on the end of the run queue and the thread at the front gets to run (see Figure 5.19). To give favored treatment to interactive threads — those that are performing the short operations of interactive users — we might somehow assign them high priorities and modify our queue so that high-priority threads are chosen before

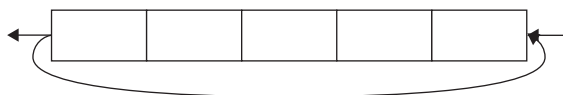
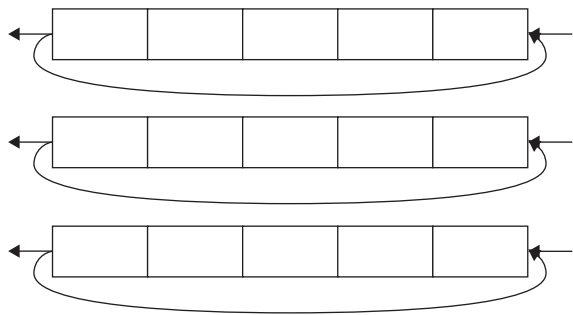
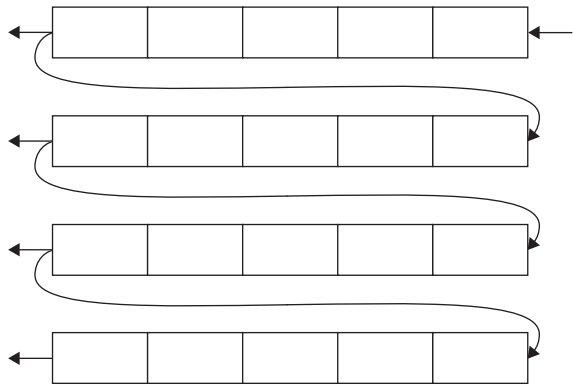


FIGURE 5.19 A round-robin queue.

**FIGURE 5.20** Round-robin queues of multiple priorities.



**FIGURE 5.21** Multi-level feedback queue.



low-priority ones. This could be done by ordering the queue according to priority, or by having multiple queues, one for each priority level (Figure 5.20).

Of course, if computer users are asked to assign the priorities themselves, every thread will be of the highest priority. Thus we need some means for automatically determining “interactiveness” and assigning appropriate priorities. Since threads requiring short bursts of processor time are more likely to be considered interactive than ones requiring long bursts, it makes sense to give the former higher priorities. But how can we determine in advance how big a burst is required? We probably can’t, without relying on the honesty of users.

Instead, let’s reduce the priority of a thread as it uses more and more time quanta. All threads run at a high priority for the first time quantum of any burst of computation. After each quantum ends, if more processor time is required, the priority gets worse. This can be implemented using a *multilevel feedback queue*, as shown in Figure 5.21. A thread becoming runnable starts at the highest priority and waits on the top-priority queue. Each time it completes a time slice it rejoins the multilevel feedback queue at the next lower queue. Threads in lower queues are not allowed to run unless there are no threads in higher queues.

This general approach makes sense, but it requires a bit more work to be usable. It’s based on the implicit assumption that our threads are idle for appreciable periods between bursts of computation. But a thread that we’d like to consider non-interactive might have a large number of relatively brief bursts of computation interspersed with short waits for disk access. Thus the length of the bursts of computation should not be the sole factor in the equation; we need to consider the time between bursts as well.

So, let’s modify the multilevel feedback queue by having threads enter the queue at a priority that depends upon what they were doing since they last were in the queue. The priority might be proportional to how long the thread was waiting (for example, for an I/O operation to complete) before returning to the run queue. This approach is conceptually simple and can be

roughly summed up by saying that a thread's priority gets worse while it's running and better while it's not. This is the basis of pretty much all thread schedulers employed on today's personal-computer and workstation operating systems, in particular Unix and Windows.

At this point we step back and note that we're using the term "priority" in a rather narrow sense. A thread's priority relates to when it will be scheduled on the processor for its next burst of execution. This is probably not what you have in mind, though, when you use the term "priority." What you probably mean relates to importance: a high-priority task is more important than, and thus, everything else being equal, should be *completed* before a lower-priority task. Thus even in the narrow context of describing operating-system schedulers, a high-priority thread should be given preferential access to resources over lower-priority threads, not just for its next request but at all times.

The user documentation of many operating systems uses this latter sense (importance) when describing thread priorities as seen by user applications, but uses the former sense (short-term scheduling order) when discussing internal priorities. Unix systems provide the *nice* command to modify the importance of a thread, so called because it's generally used to reduce the importance of a thread or process — thus one uses it to be "nice" to everyone else on the system. On Windows, one runs a thread at one of six *base priorities*, defining its importance. These base priorities are a major, but not the sole factor in determining short-term scheduling order.

#### 5.3.1.4 Shared Servers

A problem with the time-sharing approach to scheduling is that the more threads a computation uses, the greater the fraction of available processor time it gets. This is not a big deal on a personal computer, but it is a big deal on a server. Suppose that you and four friends each contribute \$1000 to buy a server. You'd probably feel that you own one-fifth of that server and thus when you run a program on it, you should get (at least) one-fifth of the processor's time. However, with the time-sharing schedulers discussed above, if you're running a single-threaded application and each of your friends are running five-threaded applications, their applications will get 20/21 of the processor's time and you will get 1/21 of it. What you (though not necessarily your friends) would like is a partitioned server in which each of you is guaranteed 20% of the server's processing time.

To accomplish such partitioning, we must account for time in terms of the user or application rather than the thread. The general concept is known as *proportional-share* scheduling — everyone gets their fair share of the computer. One interesting approach for this is *lottery scheduling* (Waldspurger and Weihl 1994), in which each user is given a certain number of lottery tickets, depending on the size of her or his share of the computer. In our example, you and each of your friends would be given one-fifth of the lottery tickets. You would give these tickets to your single thread; your friends would distribute their tickets among all their threads. Whenever the scheduler must make a scheduling decision, it essentially runs a lottery in which one lottery ticket is chosen randomly and the thread holding that ticket gets to run. Thus your thread, holding one-fifth of the tickets, is five times as likely to win as any of your friends' threads, which each hold one-twenty-fifth of the tickets.

A deterministic approach with properties similar to those of lottery scheduling is *stride scheduling* (Waldspurger and Weihl 1995). We explain it here, using somewhat different terminology. (Waldspurger and Weihl 1995) use *stride* to mean what we call the *meter rate* below. We start by assuming that we are giving fair treatment to individual threads, and that all threads are equal. Furthermore, let's assume that all threads are created when the system starts, no threads terminate, and no threads block for any reason. We'll relax all these assumptions soon.

To ensure that each thread gets its fair share of the processor, we give each thread a processor meter (rather like an electric meter) that runs, measuring processor time, only when the thread is in the run state — i.e., the thread is running. Time is measured in arbitrary units that we simply call quanta. The scheduler is driven by clock interrupts, which occur every quantum. The interrupt handler chooses the next thread to run, which is the thread with the smallest processor time on its meter. In case of tie, the thread with the lowest ID wins.

No thread will get  $i+1$  quanta of time on its meter until all other threads have  $i$  quanta on their meters. Thus with respect to the size of the quantum, the scheduler is as fair as is possible.

Let's now allow some threads to be more important than others. To become important, a thread pays a bribe to have its processor meter "fixed." To facilitate such bribes, the provider of meters has established the *ticket* as the bribery unit. It costs one ticket to obtain a "fair" meter — one that accurately measures processor time. If a thread pays two tickets, it gets a meter that measures processor time at half the rate that a fair meter does. If a thread pays three tickets, it gets a meter that measures processor time at one-third the rate of a fair meter, and so forth. Thus the rate at which a thread's meter measures processor time is inversely proportional to the number of tickets used to purchase the meter.

We make no changes to the scheduling algorithm, other than allowing threads to purchase crooked meters. Thus it is still the case that no thread will get  $i+1$  quanta of time on its meter until all other threads have  $i$  quanta on their meters, but some threads will consume more actual processor time than others. If two threads' meters have the same value, but one thread has paid  $n$  tickets for its meter and the other has paid one ticket, then the first thread will have consumed  $n$  times more processor time than the other.<sup>7</sup>

Figure 5.22 shows some of the details of slide scheduling in terms of C code. We store with each thread the bribe it has paid (in units of tickets), the meter rate induced by this bribe (*meter\_rate*), and the current meter reading (*metered\_time*). The meter is initialized with the reciprocal of the bribe, which is the amount added to the meter after each quantum of execution time. The figure also shows how the meter is updated at the end of each quantum, when the next thread is selected for execution. Note that a real implementation of the scheduler would probably use scaled-integer arithmetic, not floating-point arithmetic.

We don't show the implementation of the run queue in Figure 5.22, but it is clearly critical. If it is a balanced searched tree, where the threads are sorted by *metered\_time*, then the operations of *InsertQueue* and *PullSmallestThreadFromQueue* are done in  $O(\log(n))$  time, where  $n$  is the number of runnable threads. Though, as discussed in Section 5.3.3, many schedulers have run queues with linear-time operations, this is certainly acceptable, particularly since the number of runnable threads is not likely to be large.

```
typedef struct {
    ...
    float bribe, meter_rate, metered_time;
} thread_t;

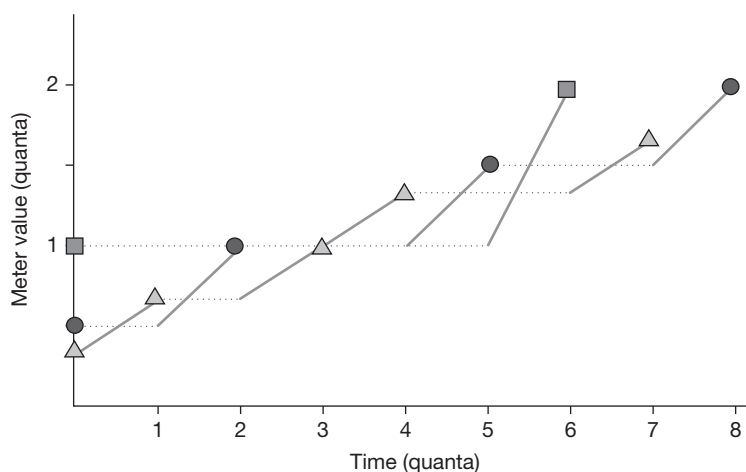
void thread_init(thread_t *t, float bribe) {
    ...
    if (bribe < 1)
        abort();
    t->bribe = bribe;
    t->meter_rate = t->metered_time = 1/bribe;
    InsertQueue(t);
}

void OnClockTick() {
    thread_t *NextThread;

    CurrentThread->metered_time +=
        CurrentThread->meter_rate;
    InsertQueue(CurrentThread);
    NextThread = PullSmallestThreadFromQueue();
    if (NextThread != CurrentThread)
        SwitchTo(NextThread);
}
```

**FIGURE 5.22** C code showing the slide-scheduler initialization required for each thread, as well as how the current thread's meter is updated at the end of a time quantum.

<sup>7</sup>Rather than normal threads paying one ticket for their meters, it is more useful for normal threads to pay, say, ten tickets for their meters. This allows not only smaller jumps in processor shares but also provides a means for giving some threads less than the normal processor share.



**FIGURE 5.23** The execution of three threads using stride scheduling. Thread 1 (a triangle) has paid a bribe of three tickets. Thread 2 (a circle) has paid two tickets, and thread 3 (a square) has paid only one ticket. The solid thick lines indicate when a thread is running. Their slopes are proportional to the meter rates (inversely proportional to the bribe).

An example of the scheduler in operation, adapted from (Waldspurger and Weihl 1995), is shown in Figure 5.23, where we are scheduling three threads that have paid three, two, and one tickets, respectively. Note that the threads' meters can differ from one another by up to one quantum.

Suppose a new thread is created. It pays its bribe and gets a meter, but to what value should the meter be initialized? If it's set to zero, then the new thread gets all the processor time until its meter catches up with the others' meters.

To deal with this problem, we figure out what value the new thread's meter would have had if the thread had been runnable or running since the beginning of time, then set the meter to this value. Thus the thread would join the run queue with no advantage (or disadvantage) over others.

How do we determine this meter value? We could look at the meter of some other thread, but its meter could differ from the desired value by up to an entire quantum. Instead, let's hypothesize a fictitious additional processor as well as a fictitious additional thread that runs only on the additional processor. This thread has as many tickets as all the (real) runnable and running threads put together and, of course, a meter that measures time in steps that are the reciprocal of this total number of tickets. It gets all the processor time of the fictitious processor, but its meter advances at the same average rate as that of any real thread that has been runnable or running since the beginning of time on the real processor. Since its meter advances more smoothly than those of the real threads, the meters of new threads are set to its value upon creation.

Implementing the fictitious thread's meter is easy — just one additional line of clock-tick code is required, as shown in Figure 5.24.

Now suppose a thread blocks, say for I/O or to wait on a semaphore. When it resumes execution, unless we make some adjustments, its meter will have the same value it had when the thread stopped execution. So, like new threads, it sets its meter to the current value of the fictitious thread's meter (though see Exercise 13).

An artifact of stride scheduling, as discussed above, is that processor time is not smoothly distributed among threads that have a large imbalance of tickets. For example, suppose thread 1 has one hundred tickets, and threads 2 through 101 each have one ticket. Thread 1 will execute for one hundred quanta, then each of the other threads will execute for one quantum each, and then the cycle repeats. Though this behavior is not necessarily bad, in some situations (as when some of the one-ticket threads are handling interactive requests) it is. A better schedule might be for thread 1's execution to alternate with each of the other threads in turn, so that thread 1 runs

```

void OnClockTick() {
    thread_t *NextThread;

    FictitiousMeter += 1/TotalBribe;
    CurrentThread->metered_time +=
        CurrentThread->meter_rate;
    InsertQueue(CurrentThread);
    NextThread = PullSmallestThreadFromQueue();
    if (NextThread != CurrentThread)
        SwitchTo(NextThread);
}

```

**FIGURE 5.24** Updated clock-tick code that maintains the meter of a fictitious thread that has paid a bribe of the sum of all the bribes paid by the real threads. This meter is used to initialize the meters of new threads and to update the meters of newly awoken threads.

for one quantum, then thread 2 runs for a quantum, then thread 1 runs for a quantum, then thread 3 runs for a quantum, and so forth.

Such scheduling is performed using a variant of stride scheduling called *hierarchical stride scheduling*. Threads are organized into groups. In our example, thread 1 would form one group and threads 2 through 101 would form another. Each group is represented by a balanced binary search tree, with the threads as leaves. Each interior node has a meter whose rate is based on the total number of tickets held by the threads at the leaves of the subtree it is a root of. Thus, for our example, the group of one-ticket threads would be represented by a tree whose root has a meter running at 1/100 speed. The singleton group for the 100-ticket thread would also be represented by a (one-node) tree whose root has a meter running at 1/100 speed.

At the end of a quantum, the group whose root has the least time on its meter is selected to run. Within the group, the thread (leaf) with the smallest time on its meter is selected and removed from the tree (this requires rebalancing and updating the meters of each of the interior nodes on the path back to the root). When the quantum of execution for the selected thread completes, it is reinserted into its tree and the meters of its new tree ancestors are updated.

In hierarchical stride scheduling, adding another thread to a group reduces the shares of the processor given to members of other groups. In some situations this might not be desirable. For example, we might want to partition a processor into  $n$  groups, with each group getting some fixed percentage of processor time regardless of how many threads are in it. Thus adding a new thread to a group changes the share of processor time given to threads of that group, but doesn't affect threads of other groups. We take this up in Exercise 14.

### 5.3.1.5 Real-Time Systems

Scheduling for real-time systems must be dependable. On a time-sharing system or personal computer, it's fine if interactive response is occasionally slow as long as most of the time it's very fast. On real-time systems, though, prolonged periods of faster-than-necessary response do not make up for any period of slower-than-necessary response. In a soft real-time application such as playing music, the faster-than-necessary response doesn't make the music sound any better, while the slower-than-necessary response produces some annoying noises. For a hard real-time application such as running a nuclear reactor, a slower-than-necessary response might necessitate a wide-area evacuation in which earlier quick responses become irrelevant and later quick responses become impossible.

Both Unix and Windows provide real-time scheduling support that both (rightly) characterize as insufficient for hard real-time applications. The approach taken is to extend time-sharing scheduling by adding some very high real-time-only priorities. Runnable real-time threads always

preempt the execution of other threads, even those performing important system functions such as network protocol processing and mouse and keyboard handling. This clearly provides fast response, but, as we explain below, not necessarily dependable response.

We shouldn't undervalue fast response — it's definitely important for many applications. So, before we discuss hard real time, what else can be done to improve response times and make them more dependable? Let's start our answer by listing some things that either slow response or make it less dependable.

- *Interrupt processing.* Even though real-time threads have the highest priority, interrupt handling still preempts their execution.
- *Caching and paging.* These techniques serve to make execution normally fast except for occasions when what is needed is not in the cache or in memory.
- *Resource acquisition.* Real-time threads must acquire kernel resources such as memory, buffers, etc., just like any other thread. If a mutex must be locked that's currently held by a lower-priority thread, the waiting thread must wait for the low-priority thread to make progress. This situation is known as *priority inversion*: the high-priority thread, while waiting, is essentially at the other thread's priority.

What can we do about these problems? To minimize the effects of interrupt processing, systems take advantage of the deferred-work techniques we discussed in Section 5.2.2. Where possible, much work that would ordinarily take place within interrupt handlers is deferred and done in contexts that can be preempted by real-time threads.

In general, caching in its various forms is considered so beneficial to overall speed that even systems supporting soft real-time applications use it. However, some hard real-time systems eschew hardware caches so as to insure that performance is uniform. To avoid delays due to paging (see Chapter 7), many systems (including Unix and Windows) allow applications to *pin* portions of their address spaces into primary memory. This means that these portions are kept in primary memory accessed without delays due to page faults, etc. Of course, doing this reduces the amount of memory available for others.

Priority inversion has a straightforward solution — *priority inheritance*. If a real-time thread is waiting to lock a mutex held by a lower-priority thread, the latter's priority is set to that of the real-time thread until the mutex is unlocked. Thus the lower-priority thread runs at the priority of the waiting real-time thread until the real-time thread acquires the lock on the mutex. If the lower-priority thread is itself waiting to lock another mutex, the priority of the holder of that mutex must be raised as well. This is known as *cascading inheritance*.

Let's now look at hard real-time systems. This is an important area of its own and we just scratch the surface here by examining two simple scheduling approaches. Say we have a number of chores to complete, each with a deadline and a known running time. A simple, intuitive approach is *earliest deadline first*: always complete the chore whose deadline is soonest. This certainly makes sense in everyday life. If you have a number of things to do, all else equal you should complete those things first that must be done first. You can work on longer-term projects when time permits, that is when you don't have an imminent deadline for some other project. Of course, this might well mean that your long-term projects get done at the last minute, but at least you meet your deadlines. If you don't end up having time to complete your long-term project, it's not because of poor scheduling, it's because you took on too many projects. In other words, if a successful schedule is at all possible, an earliest-deadline-first schedule will work.

However, things are not so simple if we have multiple processors, or, in the everyday case, if you are managing multiple people who do your work for you. The general multiprocessor scheduling problem is NP-complete. There is a vast literature on efficient algorithms for special cases and approximate algorithms for more general cases; we don't discuss them here, but see, for example, (Garey and Johnson 1975).

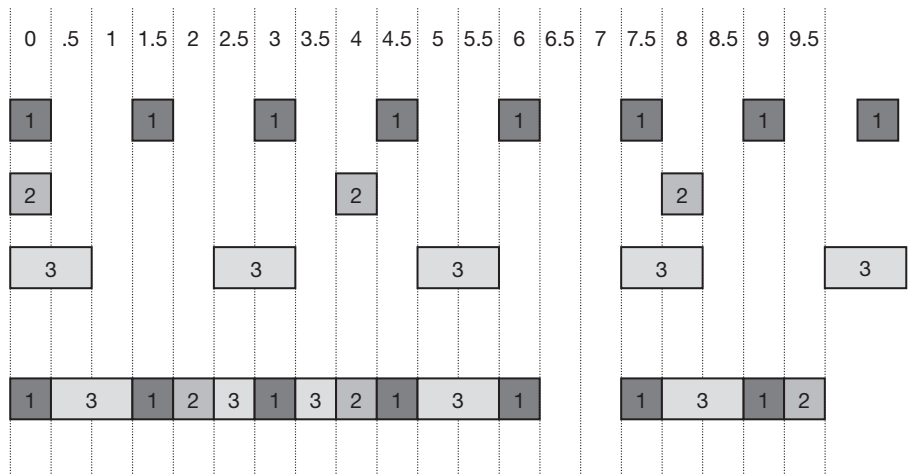
An interesting and useful single-processor case is when a number of chores must be performed periodically. Suppose we have a set of  $n$  chores such that each chore  $i$  must be completed every  $P_i$  seconds and requires  $T_i$  processing time. Of course,  $T_i$  must be less than or equal to  $P_i$ . Furthermore, the sum of the chores' duty cycles must be less than or equal to one: a chore's duty cycle is the time required for each instance divided by the length of its period — it's the fraction of the total time that must be devoted to handling this chore. If the sum of all duty cycles is greater than one, then we clearly can't do them all — there's not enough time.

If the sum of the duty cycles is less than or equal to one, then the chores can be successfully scheduled using earliest-deadline-first. However, particularly if we have a large number of chores, this scheduling algorithm is, in general, rather expensive to run: each scheduling decision requires evaluating the current status of all chores. If instead we can assign fixed priorities to the threads running the chores and use a simple preemptive priority-based scheduler, scheduling will be quick and efficient.

An intuitively attractive approach is to give threads whose chores have short periods higher priority than threads whose chores have long periods. Thus if thread  $T_i$  is handling chore  $i$ , its priority is  $1/P_i$ . The high-frequency (short-period) threads have more frequent deadlines than the low-frequency (long-period) ones and thus would seem to need the processor more often. This approach is known as *rate-monotonic scheduling* and is particularly attractive because most general-purpose operating systems provide schedulers that can handle it.

The example of this approach in Figure 5.25 shows the schedule for the first 9.5 seconds. During this period, all chores are scheduled before their deadlines. But will this continue to be so if we look beyond the first 9.5 seconds? If all chores start in phase, that is, all periods start at the same time, the answer is yes, in fact, we could have stopped after 2.5 seconds — the period of the longest-period chore. In other words, if a chore will ever fail to meet its deadline, it will fail in its first period.

To see this, consider the following. The highest-frequency (and thus highest-priority) chore runs whenever it's ready. Thus if its duty cycle is less than one, it will be successfully scheduled. The second-highest-frequency chore runs whenever both it is ready and the highest-frequency chore is not running. It's of course necessary that the sum of its duty cycle and that of the first chore be less than or equal to one, but it is not sufficient — the first chore, because of



**FIGURE 5.25** A successful application of rate-monotonic scheduling. The top three rows show three cyclic chores. The first occurs every 1.5 seconds and requires .5 seconds. The second occurs every 4 seconds and requires .5 seconds. The third occurs every 2.5 seconds and requires 1 second. The fourth row shows the schedule obtained using rate-monotonic scheduling.

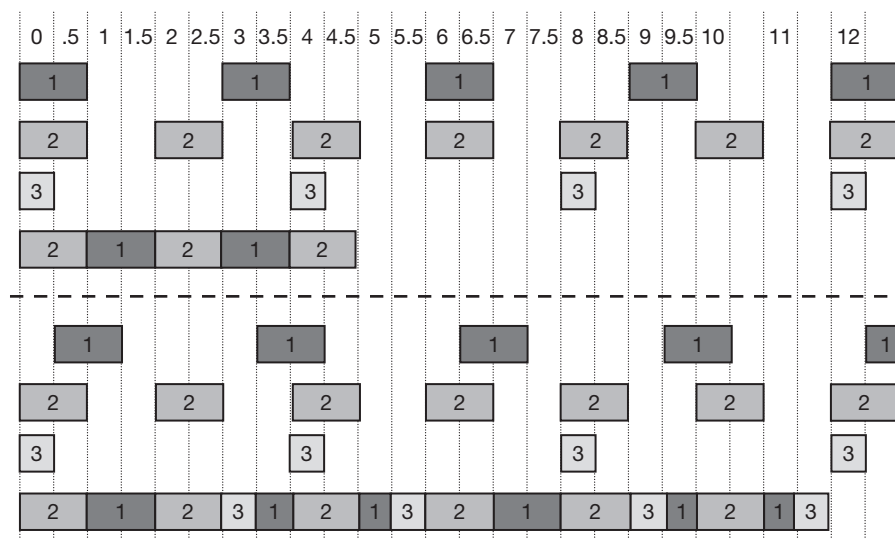
its priority, might preempt the second at a time when the second must run to meet its deadline but the first chore could wait and still meet its deadline. This would happen if, during one period of the second chore, the first used up so much time that there was not sufficient time left for the second. So, if we start the schedule so that the first chore uses the largest fraction it will ever use of the second's period, and if the second, nevertheless, is able to meet its deadline, then it will certainly be able to meet its deadline in all subsequent periods. We maximize this fraction by starting both periods at the same moment.

By applying this argument to the third-highest-frequency chore, to the fourth, and so forth, it becomes clear that all we have to consider is an initial time duration equal to the period of the lowest-frequency chore. Thus, in Figure 5.25, it's sufficient to show just the first 2.5 seconds — the period of the lowest-frequency chore.

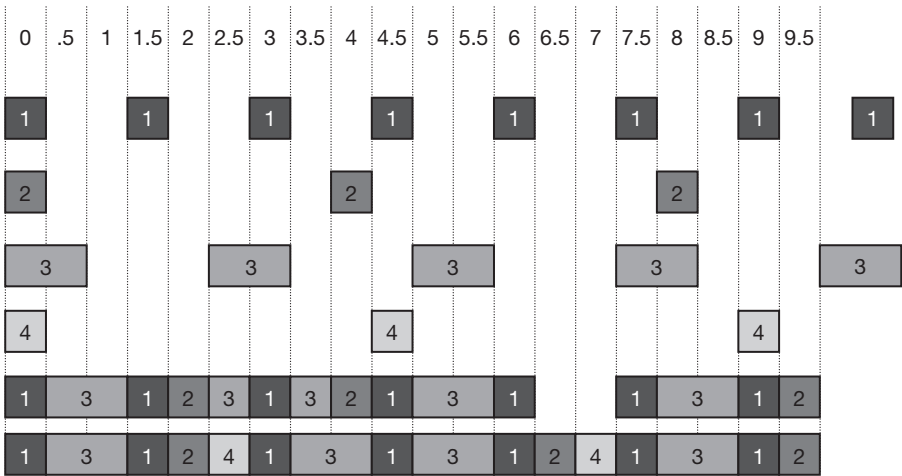
Note that the above argument applies only if the chores' periods start in phase. As shown in Figure 5.26, it might be possible, if they don't start in phase, to apply rate-monotonic scheduling successfully, even though it would fail otherwise.

Does rate-monotonic scheduling always work in the cases where the sums of the duty cycles are less than one? Figure 5.27 shows a counterexample. We add one more cyclic chore to the example of Figure 5.25, this one with a period of 4.5 seconds. With rate-monotonic scheduling, we see that the new chore cannot meet its deadline. However, as shown in the bottom line of the figure, with earliest-deadline-first scheduling all deadlines are met.

Rate-monotonic scheduling has been studied extensively in the literature, and it's been shown that no algorithm using statically assigned priorities can do better than it (Lehoczky, Sha, et al. 1989). It's also been shown (see (Lehoczky, Sha, et al. 1989) for details) that if the sum of the duty cycles is less than  $n(2^{1/n}-1)$ , where  $n$  is the number of chores, then rate-monotonic scheduling is guaranteed to work. As  $n$  gets large, this value approaches  $\ln 2$  (the natural logarithm of 2, roughly .69314718). However, this is a sufficient but not necessary condition — in Figure 5.25 the sum of the chores' duty cycles exceeds the value given by the formula, but rate-monotonic scheduling still works.



**FIGURE 5.26** The effect of phase on rate-monotonic scheduling. The top three rows show three chores. The first requires 1 second every 3 seconds, the second requires 1 second every 2 seconds, and the third requires .5 seconds every 4 seconds. The fourth row shows what happens when rate-monotonic scheduling is used: the third chore can't make its deadline even once. In the bottom half of the figure, we've started the first chore a half-second after the others. The last row shows the rate-monotonic schedule: all three chores consistently meet their deadlines.



**FIGURE 5.27** Rate-monotonic scheduling doesn’t work, but earliest-deadline-first does. We’ve added one more cyclic chore to the example in Figure 5.22, this one requiring .5 seconds every 4.5 seconds. The fifth row is the beginning of a schedule using rate-monotonic scheduling: we can’t complete the new chore within its period. However, with *earliest deadline first*, we can meet the deadline, as shown in the bottom row.

5.3.2
TACTICS

In a few special situations, techniques are needed to circumvent the scheduler’s normal actions in order to make certain that certain threads run. These situations include using local RPC (Section 4.2.2) for fast interprocess communication, synchronization on multiprocessors, and partitioning multiprocessors. We cover each of these in turn.

5.3.2.1
Handoff Scheduling

In local RPC, a thread in one process places a call to a procedure residing in another process. From the application’s point of view, the effect is as if the calling thread actually crosses the process boundary and executes code in the called process. In reality, as implemented in most operating systems, threads can do no such thing; they are restricted to executing within one process. So, two threads must be employed — one in the calling process and one in the called process.

A typical approach is that processes supplying remote procedures for others to call provide one or more threads to execute the remote procedures in response to such calls. This sounds straightforward enough: such threads wait on a queue and are woken up in response to incoming calls. The calling thread then sleeps until the called thread returns.

The problem is the scheduling latency. We’d like the time required to execute a local RPC to be not much worse than the time required to execute a strictly local procedure call. Two things are done in the RPC case that are not done in local procedure calls:

- transferring arguments and results between processes
- waking up and scheduling first the called thread at the beginning of the RPC, then the calling thread on return

We cover the former issue in Chapter 9. The problem with the latter is the time lag between when, for example, the calling thread wakes up the called thread and when the called thread is actually chosen by the scheduler to run. To eliminate this lag, we must somehow circumvent the normal actions of the scheduler and get the called thread to run immediately.

The circumvention method is called *handoff scheduling* (Black 1990a). The calling thread invokes the scheduler, passing the ID of the called thread. The calling thread blocks and the

called thread starts running immediately, using the processor originally assigned to the calling thread. For this to be reasonably fair to the other threads of the system, the scheduling state of the calling thread might also be transferred to the called thread. For example, the called thread's initial time slice is set to be whatever remains of the calling thread's time slice.

### 5.3.2.2 Preemption Control

Does it make sense to use spin locks to synchronize user threads? It certainly does not on a uniprocessor, so let's assume we are dealing with a multiprocessor. If one thread is holding a spin lock while another is waiting for it, we want to ensure that the thread holding the lock is making progress. The worst thing that could happen is for the thread holding the lock to be preempted by the thread waiting for the lock. And this is entirely possible if the time slice of the lock-holding thread expires.

The solution is somehow to convince the scheduler not to end the time slice of a thread that is holding a spin lock. In principle, this is not difficult. A thread could simply set a bit in its control structure to extend its time. This has two difficulties:

1. All threads might set this bit and never clear it. Thus the bit has no real effect.
2. The reason for using a spin lock rather than a blocking lock is performance. If setting a bit in a control structure is expensive (because the control structure belongs to the operating system), this might negate the performance benefits of using a spin lock.

To deal with the first problem, we must provide an incentive not to set the bit for longer than absolutely necessary. To deal with the second, the implementation must be extremely fast. Sun's Solaris operating system provides a mechanism that does both. If a thread executes for longer than its time slice, its scheduling priority becomes correspondingly worse. The system call that implements the operation is in the "fast track," meaning that threads calling it go into and out of the kernel with minimal overhead.

### 5.3.2.3 Multiprocessor Issues

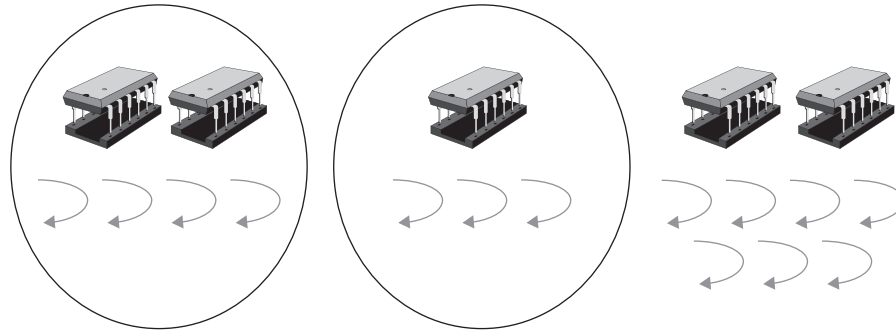
How should the processors of a multiprocessor system be scheduled? Assuming a symmetric multiprocessor (SMP) system, in which all processors can access all memory, an obvious approach is to have a single queue of runnable threads feeding all processors. Thus whenever a processor becomes idle and needs work, it takes the first thread in the queue.

This approach has been used in a number of systems but, unfortunately, suffers from two serious problems. The first is contention for the queue itself, which processors must lock before accessing. The more processors a system has, the greater the potential delay in dispatching a thread to a processor.

The second problem has to do with the caching of memory in the processors. If a thread that was running on a processor becomes runnable again, there is a good chance not only that some of the memory referenced by that thread is still in the original processor's cache, but also that the thread will reference this memory again soon. Thus it makes sense to run the thread on the same processor on which it ran previously. What a thread has brought into a processor's cache is known as its *cache footprint*. The size of the cache footprint will certainly get smaller with time while other threads are running on the processor, but as long as it still exists, there is a potential advantage to a thread's running on its most recent processor.

To deal with both these problems it makes sense to have multiple run queues, one for each processor. Thus, since each processor uses only its own queue, there is no lock contention. If a thread, when made runnable, is always put on the queue of its most recent processor, it will always take advantage of whatever cache footprint remains.

This might seem to solve all our problems, except that we need to make certain that all processors are kept reasonably busy. When a thread is created, on which processor should it run? If some processors have longer queues than others, should we attempt to balance the load?



**FIGURE 5.28** A system with two processor sets, contained in the ovals. The leftmost contains two processors and four threads; the other contains one processor and three threads. The remaining processors and threads effectively form their own set.

Load balancing will definitely cause some contention when one processor either pulls threads from or gives threads to another's run queue, but, assuming such operations are infrequent, the cost is small. The more difficult issue is the strategy for load balancing, which must take into account not only cache footprints, but also the likelihood of threads' sharing memory and how the processors and caches are organized.

In our usual model of a shared-memory multiprocessor system, any thread may run on any processor. This makes a lot of sense for most personal computers and many servers. But in many circumstances it makes sense to restrict the use of some or all of the processors. For example, a virtual-machine monitor might want to dedicate certain processors to certain virtual machines. A real-time system might need to shelter some applications from the effects of interrupt handling, so it might run their threads on processors that don't handle device interrupts.

A technique for doing this sort of partitioning, pioneered in the Mach microkernel (Black 1990b), involves the use of *processor sets*. Each such set is a collection of processors and threads. The processors may run only those threads in their set; the threads may run on only those processors in their set (see Figure 5.28). Thus for the virtual-machine example mentioned above, a virtual machine, and hence all its threads, might be assigned a processor set that's supplied by the VMM — the processors would be made available for that virtual machine. For the real-time system, critical applications might have their threads put in processor sets that include only those processors not involved with interrupt handling.

Processor sets are explicitly supported by Solaris. Windows has a similar concept that it calls affinity masks.

### 5.3.3 CASE STUDIES

In this section we examine how scheduling is done in two systems: Linux and Windows. Each has to deal with the following concerns and each does so differently.

- *Efficiency and scaling.* Scheduling decisions — which thread to run next — are made often and thus must be done efficiently. What might be practical for a personal computer with rarely more than two runnable threads doesn't necessarily work well for a busy server with many tens of runnable threads.
- *Multiprocessor issues.* It matters which processor a thread runs on. Processors have caches and if a thread has run recently on a processor, it may make sense for the thread to run on that processor again to take advantage of its "cache footprint" — those storage items it needs to use that might still be in the cache and thus accessed quickly. Another issue is that multiple

processors might cause contention on mutexes used to protect run queues and other scheduler data structures, thus slowing down scheduling.

- *Who's next?* Both systems must somehow determine which threads should be the ones that are running at any particular moment. This must take into account the relative importance of individual threads, the resources tied up by threads, and I/O performance.

### 5.3.3.1 Scheduling in Linux

Until early 2003, Linux employed a rather simple but unscalable approach to scheduling. This was replaced by a new scheduler that was not only more scalable but also more suitable for multiprocessors. The new scheduler was itself replaced in 2007 by an even newer scheduler based on stride scheduling (see Section 5.3.1.4). The general approach in the first two schedulers is to divide time into variable-length cycles and give runnable threads time on the processor each cycle roughly in proportion to their priority and in inverse proportion to how long they've been running recently. Real-time threads, however, compete only with each other on a strict priority basis: lower-priority real-time threads run only when no higher-priority threads are runnable.

Any one thread is governed by one of three scheduling policies, settable by user code. The `SCHED_FIFO` policy provides soft-real-time scheduling with high priorities and no time slicing: threads run until they terminate, block for some reason, or are preempted by a thread of even higher priority. The `SCHED_RR` policy provides soft-real-time scheduling that is just like `SCHED_FIFO` except time slicing is done using user-adjustable time quanta. The imaginatively named `SCHED_OTHER` policy provides normal time-sharing scheduling and is used by most threads.

In the old scheduler, each thread is assigned a priority as an indication of its importance. For time-sharing threads this priority is based on the thread's "nice" value (see Section 5.3.1.3). For real-time threads it's the thread's priority relative to other real-time threads, but higher than that of any time-sharing thread. For time-sharing threads, this priority is used to initialize the thread's *counter*, a variable that measures how much processor use the thread has had recently and also indicates how much of the processor it should get soon. The next thread to run depends on the result of a per-thread "goodness" computation: for real-time threads, this goodness value is based on its priority, and for time-shared threads it's the thread's counter value.

Every 10 milliseconds a clock interrupt occurs and the value of the counter for the currently running thread is decremented by one. When a (time-sharing) thread's counter becomes zero, its time slice is over and it goes back to the run queue. Thus the length of the scheduling cycle is the sum of the counters of all the runnable threads. At the end of a scheduling cycle, when there are no runnable real-time threads and all runnable time-sharing threads have zero counters, the counters of all time-sharing threads, not just the runnable ones, are set as follows:

```
counter = counter/2 + priority;
```

Thus the counters for the runnable threads are reset to the threads' priorities ("nice" values), while those of sleeping threads increase to a maximum of twice the threads' priorities. Threads that have been sleeping for a while end up with a large share of the scheduling cycle the next time they run. Since such sleeping threads are likely to be interactive threads — they may have been sleeping or waiting for the next keystroke or mouse click — interactive threads get favored treatment for their next burst of processor usage.

What's wrong with this old scheduler? Why was it replaced? There are a number of problems. Determining the next thread to run requires computing the goodness value for all threads on the run queue — it's the first one encountered with the highest goodness value. Thus performance suffers with larger run queues. What's equally bad, if not worse, is that the counter values of all time-sharing threads, not just all runnable ones, must be recomputed at the end of each scheduling cycle. For a server with thousands of threads, this can be time-consuming.

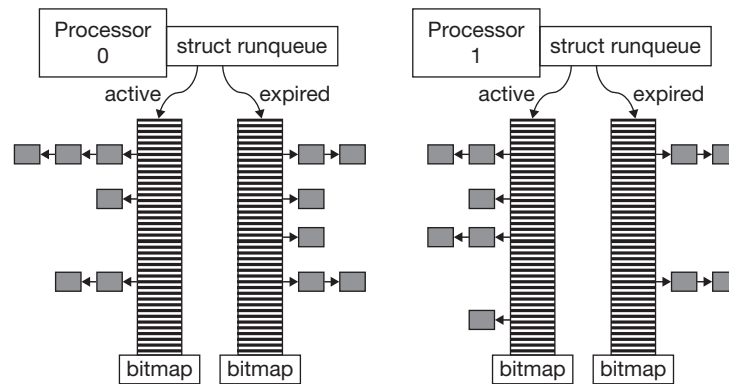


FIGURE 5.29 The run queues of the  $O(1)$  Linux scheduler.

In addition, there is no explicit support for multiprocessors. On such a system, the one run queue serves all processors; a thread is equally likely to run on any processor from one execution to the next. Furthermore, with a single run queue there is contention for the mutex protecting it.

The new scheduler, known as the  $O(1)$  scheduler for reasons explained below, has a roughly similar effect to the old one in determining which thread runs when, but does so more efficiently and takes cache footprints into account when scheduling for multiprocessors.

Each processor has a separate run queue — actually a separate pair of run queues labeled *active* and *expired* (see Figure 5.29). Each run queue is itself an array of queues, one for each priority level, of which there are 140. Attached to each run queue is a bit vector indicating which of the queues are non-empty. Finding the highest-priority runnable thread involves searching the bit vector for the first non-empty queue, then taking the first thread from that queue. Thus scheduling decisions are made in constant time, as opposed to the linear time required by the old scheduler — thus explaining the name of the scheduler.

A processor's active queue provides it with threads to run. When one is needed, the thread from the front of the highest-priority non-empty queue is chosen and runs with a time slice that depends on its priority. When a thread's time slice is over, what happens next also depends on its priority. Real-time threads (necessarily `SCHED_RR` since `SCHED_FIFO` threads aren't time-sliced) go back to the active queue at their priority. A time-sharing thread's priority is reduced; if it's still above an interactive-priority threshold, it goes back to the active queue at its new priority. Otherwise it goes to the expired queue. However, if threads have been waiting in the expired queue for too long (how long depends on how many there are), then all time-sharing threads go to the expired queue when their time slice is over.

If there are no threads in the active queue, which means there are no runnable real-time threads, then the active and expired queues are switched. The threads that were on the expired queue now compete for the processor.

When a thread that has been sleeping wakes up, it's assigned a priority that depends both on how long it was sleeping and what it was waiting for. The longer the sleep, the better its priority becomes. If it was waiting on a hardware event such as a keystroke or a mouse click, its priority becomes even better. The assumption is that long-term sleepers or those who had been waiting for such events are the most likely to be interactive threads. Newly awoken threads go on the active queue.

The effect of all this is that real-time threads run to the exclusion of all other threads. Threads determined to be interactive get favored treatment over non-interactive threads.

As we've mentioned, each processor has its own set of queues. Threads typically run on the same processor all the time, thus taking advantage of their cache footprints. Of course, we also need a means for sharing the workload among all processors — the benefits of using the cache footprint do not outweigh those of using multiple processors.

What the  $O(1)$  scheduler does is to have a clock interrupt on each processor every millisecond. If the interrupt handler sees the processor's run queues are empty, it finds the processor with the largest load and steals threads from it (assuming not all processors are idle). Every 250 milliseconds the interrupt handler checks for a load imbalance — if its processor's run queue is much smaller than others, it also steals threads from the others.

The result of all this is threefold:

- Threads rarely migrate from one processor to another — thus advantage is taken of cache footprints.
- Queues remain in balance over the long term.
- Processors rarely access one another's queues and thus lock contention is rare.

**The Completely Fair Scheduler** Despite the improvements gained with the  $O(1)$  scheduler, the Linux developers decided, apparently because of a few examples of anomalous behavior, to replace it with a scheduler based on stride scheduling (see Section 5.3.1.4) and called the *completely fair scheduler* (CFS). (The CFS approach was apparently developed without knowledge of the prior work on stride scheduling, which was described twelve years earlier (Waldspurger and Weihl 1995) — there is no mention of stride scheduling in any of the CFS documentation. Since stride scheduling requires logarithmic time, CFS might be called the  $O(\log(n))$  scheduler.)

In support of CFS, the scheduler architecture was changed to allow the use of a number of scheduling policies. Standard Linux supplies two: a real-time policy supporting the POSIX `SCHED_RR` and `SCHED_FIFO` classes and a fair policy (stride scheduling) supporting the other POSIX scheduling classes. The policies are ordered so that no runnable threads in a lower policy are scheduled if there are any runnable threads in a higher policy. Thus when the scheduler makes a decision, it first invokes the real-time policy to select a thread; then, if no real-time threads are available to run, it invokes the fair policy. The real-time policy is implemented much as it was in the  $O(1)$  scheduler, but without the expired queue.

Threads of all scheduling policies are assigned to individual processors and scheduling is done separately for each processor. Just as in the  $O(1)$  scheduler, load balancing is done to even out the number of threads assigned to each processor.

### 5.3.3.2 Scheduling in Windows

The Windows scheduler is essentially round-robin with multiple priorities, but with a few twists. Its basic strategy is straightforward. Threads are assigned priorities ranging from 0 through 31, with 0 reserved for special idle threads. Priorities of normal threads must be less than 16 and greater than 0; “real-time” threads have priorities from 16 through 31. Normal threads are assigned a fixed base priority, but their effective priority is “boosted” when they wake up after sleeping and is reduced while they are running. The priorities of real-time threads are fixed. Users assign base priorities to threads according to their importance.

Another scheduling parameter is the length of a thread's time quantum — how long it runs until preempted by a thread of equal priority. Normal threads are assigned a default quantum whose value depends on the type of system. Servers typically have longer quanta than interactive computers. But subsystems external to the actual scheduler can change the quanta of individual threads while they are running. In particular, the Win-32 subsystem, which manages windows on the display, increases the quanta of foreground threads — threads belonging to the process of the foreground window. The effect of doing this is to make sure that these threads get a greater portion of the processor's time than do threads of equal priority belonging to background processes. Note that simply assigning such threads a higher priority might prevent background threads from running at all — if that is what is desired, then the user can give such threads a lower base priority.

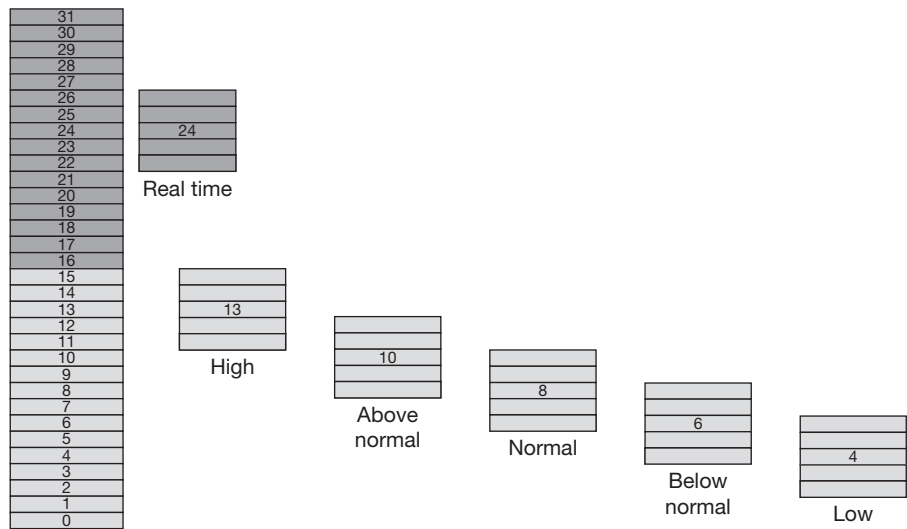


FIGURE 5.30 Priority ranges in Windows.

The base priorities assigned to threads are typically from the middle of ranges labeled *high*, *above normal*, *normal*, *below normal*, and *low*, as shown in Figure 5.30. Real-time threads are typically assigned priorities from the range labeled *real time*, though their priorities do not change. A normal thread’s effective priority is some value equal to or greater than its base. When waking up from a sleep, a thread’s effective priority is set to their base priority plus some wait-specific value (usually in the range 1 to 6, depending on what sort of event it was waiting for). A thread’s effective priority is decremented by one, but to no less than the base, each time a quantum expires on it.

The effect of all this is that threads of the same range share the processor with one another, but threads of lower ranges cannot run at all unless there are no runnable threads in higher ranges.

As described so far, the Windows scheduler handles typical interactive computing and servers reasonably well, but, despite its real-time priorities, it doesn’t handle many real-time chores very well. One issue is priority inversion, as described in Section 5.3.1.5. Rather than attempt to detect instances of priority inversion, Windows makes sure that all runnable processes eventually make progress (though threads running at real-time priorities can prevent other threads from running). A system thread known as the *balance set manager*, whose primary job is to assist in managing virtual memory, periodically checks for threads that have been runnable for a certain period of time, but have not actually been running. It increases their priority to 15, the maximum value for normal threads. Once such a thread has run for its time quantum, its priority goes back to what it was.

Another issue is handling applications, such as audio and video, that have rather stringent performance requirements and, if given high real-time priorities, could monopolize the processors. A system running only such applications can be scheduled using rate-monotonic scheduling (Section 5.3.1.5), but such scheduling doesn’t take into account the requirements of non-periodic “normal” applications.

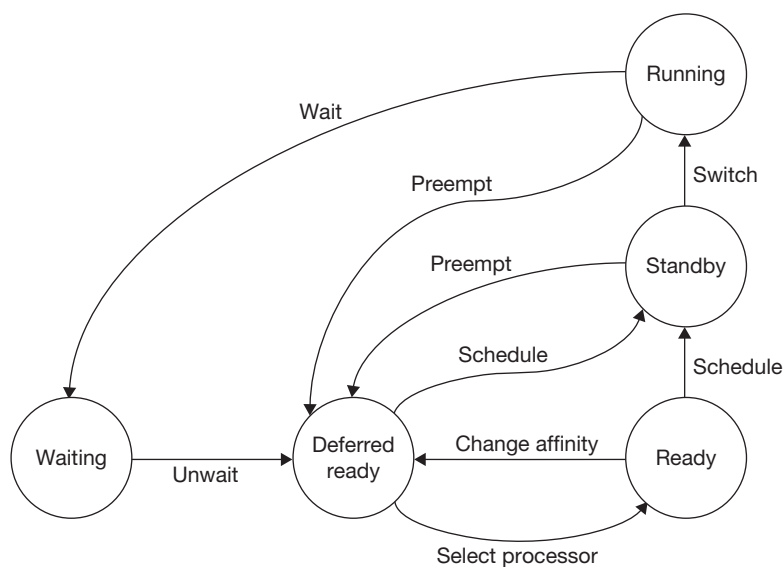
Windows, beginning with Windows Vista, handles this with an approach called the *multimedia class scheduler service* (MMCSS) in which thread priorities are dynamically adjusted so that they can meet their constraints without monopolizing the processors. Threads needing this service are called *multimedia threads*. They register for the service by indicating at what real-time priority they should run and how much processor time should be reserved for other (normal)

activity — by default, 20%. The service is provided by user threads, running at a high real-time priority (27), that monitor the multimedia threads and boost their priority to the desired range for (by default) 80% of the time, but lower their priorities to the low normal range for 20% of the time. Thus, over a 10-millisecond period, their priorities are in the real-time range for 8 milliseconds, but drop to low values for 2 milliseconds.<sup>8</sup>

Unfortunately, this by itself is not quite enough to guarantee that the multimedia threads get 80% of processor time. Recall that Windows uses deferred procedure calls (Section 5.2.2) to cope with such potentially time-consuming activity as handling network traffic. Since this work takes place in the interrupt context, it preempts the execution of threads, even real-time threads. Thus, despite MMCSS, the multimedia threads may suffer because of network activity. This could perhaps be dealt with by doing network-protocol processing by kernel threads rather than DPCs (though see Exercise 17); Windows handles it by having MMCSS direct the network-protocol code (running as DPCs) to “throttle back” and thus reduce the rate at which network packets are handled.

The Windows scheduler is implemented using an elaborate set of states (Figure 5.31) and queues. Associated with each processor is a set of *ready queues*, one per scheduling priority level. These queues contain threads that are to run on the given processor. In addition, each processor has a *deferred ready queue*, containing runnable threads that have not yet been assigned to a particular processor. There are any number of queues of threads waiting for some sort of event to occur.

To see how this works, let’s follow the life of a thread. When it’s created and first made runnable, its creator (running in kernel mode) puts it in the deferred ready state and enqueues it in the deferred ready queue associated with the current processor. It’s also randomly assigned an *ideal processor*, on which it will be scheduled if available. This helps with load balancing. Its creator (or, later, the thread itself) may also give it an *affinity mask* (Section 5.3.2.3) indicating the set of processors on which it may run.



**FIGURE 5.31** Scheduler states in Windows.

<sup>8</sup>This description is based on <http://technet.microsoft.com/en-us/magazine/cc162494.aspx>.

Each processor, each time it completes the handling of the pending DPC requests, checks its deferred ready queue. If there are any threads in it, it processes them, assigning them to processors. This works as follows. The DPC handler first checks to see if there are any idle processors that the thread can run on (if it has an affinity mask, then it can run only on the indicated processors). If there are any acceptable idle processors, preference is given first to the thread's ideal processor, then to the last processor on which it ran (to take advantage of the thread's cache footprint (Section 5.3.2.3)). The thread is then put in the *standby* state and given to the selected processor as its next thread. The processor would be currently running its idle thread, which repeatedly checks for a standby thread. Once found, the processor switches to the standby thread.

If there are no acceptable idle processors, then the thread is assigned to its ideal processor. The DPC handler checks to see if the thread has a higher priority than what's running on that processor. If so, it puts the thread in the standby state, and sends the processor an interrupt. When the processor returns from its interrupt handler it will notice the higher-priority thread in standby state and switch to it, after first putting its current thread on its deferred ready list. Otherwise, the DPC handler puts the thread in the ready state and puts it in one of the ideal processor's ready queues according to the thread's priority.

When a thread completes its time quantum (which, as discussed in Section 5.2.2, is dealt with by a DPC), the processor searches its ready queues for a thread of equal or higher priority and switches to it, if it finds one. Otherwise it continues with the current thread.

An executing thread might perform some sort of blocking operation, putting itself in a wait queue. Its processor then searches its ready queues for the next thread to run.

When a thread is made runnable after it has been in the wait state, it's put into the deferred ready queue of the processor on which the thread doing the *unwait* operation was running.

Other operations shown in Figure 5.31 include preempting a thread that's in the state (possible, though unlikely) and changing the affinity mask of a thread that's already been assigned a processor, making the current processor selection unacceptable.

## 5.4 CONCLUSIONS

Processor management entails multiplexing the available processors to handle all the activities taking place in a computer system — almost everything involves the use of one or more processors. We started our discussion by looking at implementation strategies for threads. From an application's point of view, a thread is the processor; everything having to do with threads is essential to performance. Interrupt processing, though hidden from applications, also has a performance role. By their nature, interrupts displace other activity. Thus we need careful control over when they can occur. In many cases it is important to do only what is absolutely necessary within interrupt handlers, relegating work to other contexts so it can be done with minimal disruption.

Scheduling is a topic unto itself. We briefly examined its theory, in terms of the basic strategies employed in computer systems. In practice, at least for interactive systems, a fair amount of attention is paid to determining which threads are indeed interactive, giving them favored treatment. But a major theme in all the operating systems we have looked at is scalability — the scheduler must perform well, with low overhead, on large, busy systems.

## 5.5 EXERCISES

1. Suppose you are designing a server that requires many thousands of concurrent threads. Which approach would be the most suitable: the one-level model or the two-level model with multiple kernel threads? Explain.

- \* 2. Implementing POSIX threads on Linux was made difficult because of Linux's use of variable-weight processes. Changes had to be made to the process model to make possible the more efficient NTPL implementation.
  - a. One problem was that, since Linux "threads" were actually processes, the only way to wait for the termination of a thread was via the *wait* family of system calls. Explain why this was a problem. (*Hint*: consider how *pthread\_join* can be used.)
  - b. This problem was overcome in a rather creative but not terribly efficient way. How might you handle this problem if you were implementing POSIX threads?
- 3. An *upcall* is a mechanism by which kernel code can place a call into user code — it is essentially the reverse of a system call. Explain how it might be implemented. (*Hint*: consider what resources must be available in the user process.)
- \* 4. The following code is an alternative to the implementation of mutexes given in Section 5.1.2. Does it work? Explain why or why not.

```

kmutex_lock(mutex_t *mut) {
    if (mut->locked) {
        enqueue(mut->wait_queue, CurrentThread);
        thread_switch();
    }
    mut->locked = 1;
}

kmutex_unlock(mutex_t *mut) {
    mut->locked = 0;
    if (!queue_empty(mut->wait_queue))
        enqueue(RunQueue, dequeue(mut->wait_queue));
}

```

- 5. The simple implementation of *thread\_switch* in Section 5.1.2 doesn't deal with the case of the run queue's being empty. Assuming that threads are either runnable or waiting on a mutex, what can you say about the system if there are no threads in the run queue?
- \* 6. The final implementation of *blocking\_lock* on page 174 requires some changes to *thread\_switch*. Show how *thread\_switch* must be modified.
- \* 7. Show how to implement semaphores in terms of futexes. Be sure to give the implementations of both the P and V operations.
- \* 8. We have a new architecture for interrupt handling. There are  $n$  possible sources of interrupts. A bit vector is used to mask them: if bit  $i$  is 1, then interrupt source  $i$  is masked. The operating system employs  $n$  threads to handle interrupts, one per interrupt source. When interrupt  $i$  occurs, thread  $i$  handles it and interrupt source  $i$  is automatically masked. When the thread completes handling of the interrupt, interrupt source  $i$  is unmasked. Thus if interrupt source  $i$  attempts to send an interrupt while a previous interrupt from  $i$  is being handled, the new interrupt is masked until the handling of the previous one is completed. In other words, each interrupt thread handles one interrupt at a time.

Threads are scheduled using a simple priority-based scheduler. It maintains a list of runnable threads (the exact data structure is not important for this problem). There's a global variable *CurrentThread* that refers to the currently running thread.

- a. When an interrupt occurs, on which stack should the registers of the interrupted thread be saved? Explain. (*Hint*: there are two possibilities: the stack of the interrupted thread and the stack of the interrupt-handling thread.)
  - b. After the registers are saved, what further actions are necessary so that the interrupt-handling thread and the interrupted thread can be handled by the scheduler? (*Hint*: consider the scheduler's data structures.)
  - c. Recall that Windows employs DPCs (deferred procedure calls) so that interrupt handlers may have work done when there is no other interrupt handling to be done. How could this be done in the new architecture? (*Hint*: it's easily handled.)
  - d. If there are multiple threads at the same priority, we'd like their execution to be time-sliced — each runs for a certain period of time, then yields to the next. In Windows, this is done by the clock interrupt handler's requesting a DPC, which forces the current thread to yield the processor. Explain how such time-slicing can be done on the new architecture.
9. Consider the implementation of DPCs given in Section 5.2.2. The intent of the DPC mechanism is to deal with chores generated by interrupt handlers at a lower priority level and thus not interfere with higher-priority interrupts. This mechanism works well; however, if there are a lot of these chores, it could prevent equally important threads from running.
- a. Describe the existing mechanism that ensures that DPC requests are handled in preference to threads. (*Hint*: this is easy.)
  - b. Describe (i.e., invent) a mechanism to solve this problem. That is, how can we limit the number of DPC requests that are processed in preference to normal thread execution such that the remaining DPC requests are processed on an equal basis with normal threads? (*Hint*: this is slightly less easy.)
10. Explain why Windows deferred procedure calls (DPCs) may not access user memory, but asynchronous procedure calls (APCs) may.
- \*11. An operating system has a simple round-robin scheduler used in conjunction with time slicing: when a thread's time slice is over, it goes to the end of the run queue and the next thread runs. The run queue is implemented as a singly linked list of threads, with pointers to the first and last threads in the queue. Assume for parts a and b that we have a uniprocessor system.
- a. The system has a mix of long-running compute threads that rarely block and interactive threads that spend most of their time blocked, waiting for keyboard input, then have very brief bursts of using the processor. Assuming we want the system to have good interactive response, explain what is wrong with the scheduler.
  - b. How might the scheduler be improved to provide good interactive response? (*Hint*: a simple improvement is sufficient.)
  - c. We add three more processors to our system and add the appropriate synchronization (spin locks) to our scheduler data structures. Describe the performance problems that will arise.
  - d. Describe what might be done to alleviate these performance problems, yet still have reasonable parallelism.
- \*12. Explain why the APC interrupt priority level must be lower than that of a DPC.
- \*13. Figure 5.24 shows how threads' meters are updated after each clock tick under stride scheduling. *FictitiousMeter* is used to initialize the meters of new threads and of threads rejoining the run queue after having been sleeping. However, when a thread blocks, its meter's value probably is not the same as that of *FictitiousMeter*.

- a. Explain why this is so.
  - b. Why might it be reasonable to keep track of this difference between the thread's meter value and that of *FictitiousMeter* and to add this difference to the current value of *FictitiousMeter* when the thread rejoins the run queue?
- \*14. In hierarchical stride scheduling, whenever a new thread joins a group, the total number of tickets held by the group increases and thus so does that group's collective share of processor time. A better approach might be to give each group a fixed number of tickets to be evenly distributed among all its members. However, it might be a bit time-consuming to readjust each member thread's bribe whenever a new thread joins the group. Describe how we might modify hierarchical stride scheduling so that each group's share of processor time remains constant despite the addition or deletion of group members, and that such addition and deletion is done in constant time (not counting the time to update the balanced tree).
15. Suppose, in the scheduling scenario in Figure 5.25, each cyclic chore is handled by a thread — one thread per cyclic chore. Show how the scheduling constraints can be satisfied on either Unix or Windows. Note that there are actually two constraints: each chore must finish exactly once per cycle and each chore must start exactly once per cycle. The first constraint is handled by the scheduler, the second by the thread itself (perhaps by waiting on a timer).
16. Why does the Linux  $O(1)$  scheduler maintain two queues per processor — the active queue and the expired queue? (*Hint*: consider load balancing and cache footprints.)
- \*17. Windows performs network-protocol processing in the interrupt context using DPCs. As explained in Section 5.3.3.2, this can cause interference with multimedia applications that, despite running at real-time priorities, are preempted by network-protocol processing. An alternative approach might be to have special kernel threads handle the network-protocol processing and thus do it under the control of the scheduler, which could then give favored treatment to multimedia applications. Explain what the disadvantage of this approach would be. (*Hint*: consider things from the point of view of network performance, even without multimedia applications.)
18. Explain why the Windows scheduler has the *standby* state, rather than simply having a processor run the highest priority thread in its *ready* queues.

Anderson, T. E., B. N. Bershad, E. Lazowska, H. M. Levy (1992). Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems* **10**(1): 53–79.

Aral, Z., J. Bloom, T. W. Doepfner, I. Gertner, A. Langerman, G. Schaffer (1989). Variable-Weight Processes with Flexible Shared Resources. *Proceedings of the Winter 1989 USENIX Technical Conference*.

Black, D. L. (1990a). *Scheduling Support for Concurrency and Parallelism in the Mach Operating System*. IEEE Computer **23**(5): 35–43.

Black, D. L. (1990b). *Scheduling and Resource Management Techniques for Multiprocessors*. School of Computer Science, Carnegie Mellon University CMU Thesis CMU-CS-90-152.

Doepfner, T. W. (1987). *Threads: A System for the Support of Concurrent Programming*, Brown University, at <http://www.cs.brown.edu/~twd/ThreadsPaper.pdf>

Garey, M. R. and D. S. Johnson (1975). Complexity Results for Multiprocessor Scheduling Under Resource Constraints. *SIAM Journal of Computing* **4**(4): 392–411.

Kleiman, S. R. and J. Eykholt (1995). Interrupts as Threads. *ACM SIGOPS Operating Systems Review* **29**(2): 21–26.

Lehoczky, J., L. Sha, Y. Ding (1989). The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. *Proceedings of the Real-Time Systems Symposium*, 166–171.

## 5.6 REFERENCES

- Von Behren, R., Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer (2003). Capriccio: Scalable Threads for Internet Services. *Nineteenth Symposium on Operating Systems Principles*. Lake George, NY, ACM.
- Waldspurger, C. A. and W. E. Weihl (1994). Lottery Scheduling: Flexible Proportional-Share Resource Management. *Proceedings of the First Symposium on Operating Systems Design and Implementation*. Monterey, USENIX.
- Waldspurger, C. A. and W. E. Weihl (1995). Stride Scheduling: Deterministic Proportional-Share Resource Management. Massachusetts Institute of Technology Technical Memorandum LCS/TM-528.