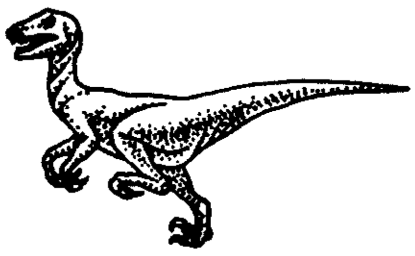
CHAPTER



CPU scheduling is the basis of multiprogrammed operating systems. By switching the CPU among processes, the operating system can make the computer more productive. In this chapter, we introduce basic CPU-scheduling concepts and present several CPU-scheduling algorithms. We also consider the problem of selecting an algorithm for a particular system.

In Chapter 4, we introduced threads to the process model. On operating systems that support them, it is kernel-level threads—not processes—that are in fact being scheduled by the operating system. However, the terms **process scheduling** and **thread scheduling** are often used interchangeably. In this chapter, we use *process scheduling* when discussing general scheduling concepts and *thread scheduling* to refer to thread-specific ideas.

5.1

In a single-processor system, only one process can run at a time; any others must wait until the CPU is free and can be rescheduled. The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The idea is relatively simple. A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU then just sits idle. All this waiting time is wasted; no useful work is accomplished. With multiprogramming, we try to use this time productively. Several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues. Every time one process has to wait, another process can take over use of the CPU.

Scheduling of this kind is a fundamental operating-system function. Almost all computer resources are scheduled before use. The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to operating-system design.

5.1.1 CPU–I/O Burst Cycle

The success of CPU scheduling depends on an observed property of processes: Process execution consists of a **cycle** of CPU execution and I/O wait. Processes

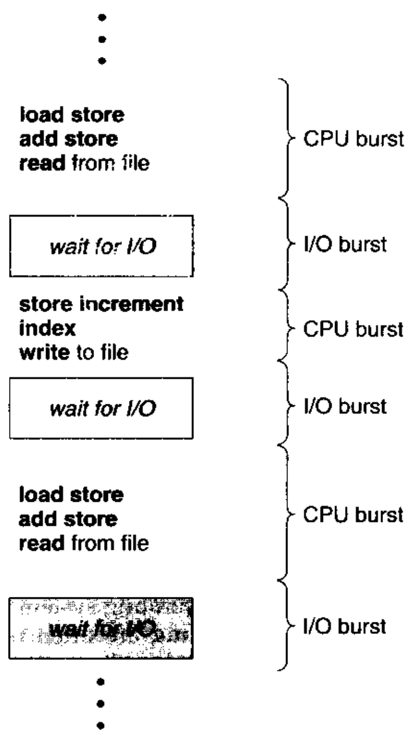


Figure 5.1 Alternating sequence of CPU and I/O bursts.

alternate between these two states. Process execution begins with a **CPU burst**. That is followed by an **I/O burst**, which is followed by another CPU burst, then another I/O burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution (Figure 5.1).

The durations of CPU bursts have been measured extensively. Although they vary greatly from process to process and from computer to computer, they tend to have a frequency curve similar to that shown in Figure 5.2. The curve is generally characterized as exponential or hyperexponential, with a large number of short CPU bursts and a small number of long CPU bursts. An I/O-bound program typically has many short CPU bursts. A CPU-bound program might have a few long CPU bursts. This distribution can be important in the selection of an appropriate CPU-scheduling algorithm.

5.1.2 CPU Scheduler

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the **short-term scheduler** (or CPU scheduler). The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

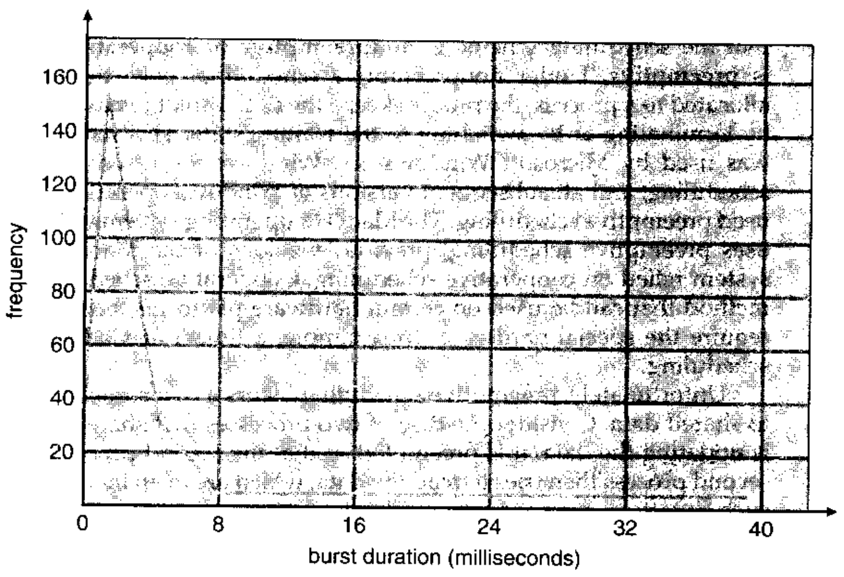


Figure 5.2 Histogram of CPU-burst durations.

Note that the ready queue is not necessarily a first-in, first-out (FIFO) queue. As we shall see when we consider the various scheduling algorithms, a ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list. Conceptually, however, all the processes in the ready queue are lined up waiting for a chance to run on the CPU. The records in the queues are generally process control blocks (PCBs) of the processes.

5.1.3 Preemptive Scheduling

CPU-scheduling decisions may take place under the following four circumstances:

When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of `wait` for the termination of one of the child processes)

When a process switches from the running state to the ready state (for example, when an interrupt occurs)

When a process switches from the waiting state to the ready state (for example, at completion of I/O)

When a process terminates

For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however, for situations 2 and 3.

When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is **nonpreemptive** or **cooperative**; otherwise, it is **preemptive**. Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state. This scheduling method was used by Microsoft Windows 3.x; Windows 95 introduced preemptive scheduling, and all subsequent versions of Windows operating systems have used preemptive scheduling. The Mac OS X operating system for the Macintosh uses preemptive scheduling; previous versions of the Macintosh operating system relied on cooperative scheduling. Cooperative scheduling is the only method that can be used on certain hardware platforms, because it does not require the special hardware (for example, a timer) needed for preemptive scheduling.

Unfortunately, preemptive scheduling incurs a cost associated with access to shared data. Consider the case of two processes that share data. While one is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state. In such situations, we need new mechanisms to coordinate access to shared data; we discuss this topic in Chapter 6.

Preemption also affects the design of the operating-system kernel. During the processing of a system call, the kernel may be busy with an activity on behalf of a process. Such activities may involve changing important kernel data (for instance, I/O queues). What happens if the process is preempted in the middle of these changes and the kernel (or the device driver) needs to read or modify the same structure? Chaos ensues. Certain operating systems, including most versions of UNIX, deal with this problem by waiting either for a system call to complete or for an I/O block to take place before doing a context switch. This scheme ensures that the kernel structure is simple, since the kernel will not preempt a process while the kernel data structures are in an inconsistent state. Unfortunately, this kernel-execution model is a poor one for supporting real-time computing and multiprocessing. These problems, and their solutions, are described in Sections 5.4 and 19.5.

Because interrupts can, by definition, occur at any time, and because they cannot always be ignored by the kernel, the sections of code affected by interrupts must be guarded from simultaneous use. The operating system needs to accept interrupts at almost all times; otherwise, input might be lost or output overwritten. So that these sections of code are not accessed concurrently by several processes, they disable interrupts at entry and reenable interrupts at exit. It is important to note that sections of code that disable interrupts do not occur very often and typically contain few instructions.

5.1.4 Dispatcher

Another component involved in the CPU-scheduling function is the **dispatcher**. The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves the following:

- Switching context
- Switching to user mode

- Jumping to the proper location in the user program to restart that program

The dispatcher should be as fast as possible, since it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**.

5.2 [illegible]

Different CPU scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.

Many criteria have been suggested for comparing CPU scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best. The criteria include the following:

- **CPU utilization**. We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).

- **Throughput**. If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called *throughput*. For long processes, this rate may be one process per hour; for short transactions, it may be 10 processes per second.

- **Turnaround time**. From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the *turnaround time*. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

- **Waiting time**. The CPU scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue. *Waiting time* is the sum of the periods spent waiting in the ready queue.

- **Response time**. In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called *response time*, is the time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time. In most cases, we optimize the average measure. However, under some circumstances, it is desirable to optimize the minimum or maximum values rather than the average. For example, to guarantee that all users get good service, we may want to minimize the maximum response time.

Investigators have suggested that, for interactive systems (such as time-sharing systems), it is more important to minimize the *variance* in the response time than to minimize the average response time. A system with reasonable and *predictable* response time may be considered more desirable than a system that is faster on the average but is highly variable. However, little work has been done on CPU-scheduling algorithms that minimize variance.

As we discuss various CPU-scheduling algorithms in the following section, we will illustrate their operation. An accurate illustration should involve many processes, each being a sequence of several hundred CPU bursts and I/O bursts. For simplicity, though, we consider only one CPU burst (in milliseconds) per process in our examples. Our measure of comparison is the average waiting time. More elaborate evaluation mechanisms are discussed in Section 5.7.

5.3 Scheduling Algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. There are many different CPU scheduling algorithms. In this section, we describe several of them.

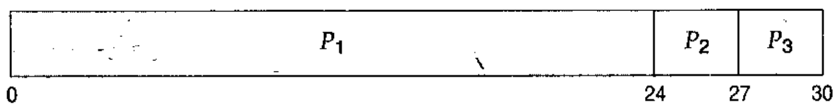
5.3.1 First-Come, First-Served Scheduling

By far the simplest CPU-scheduling algorithm is the **first-come, first-served (FCFS) scheduling algorithm**. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. The code for FCFS scheduling is simple to write and understand.

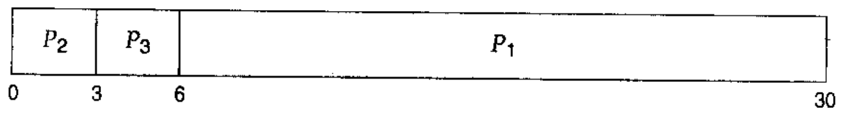
The average waiting time under the FCFS policy, however, is often quite long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

If the processes arrive in the order P_1, P_2, P_3, and are served in FCFS order, we get the result shown in the following **Gantt chart**:



The waiting time is 0 milliseconds for process P_1, 24 milliseconds for process P_2, and 27 milliseconds for process P_3. Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds. If the processes arrive in the order P_2, P_3, P_1, however, the results will be as shown in the following Gantt chart:



The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds. This reduction is substantial. Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the process's CPU burst times vary greatly.

In addition, consider the performance of FCFS scheduling in a dynamic situation. Assume we have one CPU-bound process and many I/O-bound processes. As the processes flow around the system, the following scenario may result. The CPU-bound process will get and hold the CPU. During this time, all the other processes will finish their I/O and will move into the ready queue, waiting for the CPU. While the processes wait in the ready queue, the I/O devices are idle. Eventually, the CPU-bound process finishes its CPU burst and moves to an I/O device. All the I/O-bound processes, which have short CPU bursts, execute quickly and move back to the I/O queues. At this point, the CPU sits idle. The CPU-bound process will then move back to the ready queue and be allocated the CPU. Again, all the I/O processes end up waiting in the ready queue until the CPU-bound process is done. There is a **convoy effect** as all the other processes wait for the one big process to get off the CPU. This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.

The FCFS scheduling algorithm is nonpreemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O. The FCFS algorithm is thus particularly troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals. It would be disastrous to allow one process to keep the CPU for an extended period.

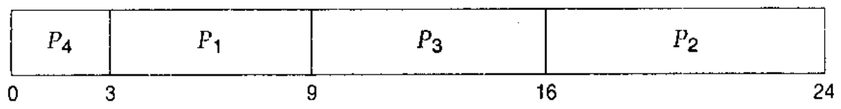
5.3.2 Shortest-Job-First Scheduling

A different approach to CPU scheduling is the **shortest-job-first (SJF) scheduling algorithm.** This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie. Note that a more appropriate term for this scheduling method would be the *shortest-next-CPU-burst algorithm,* because scheduling depends on the length of the next CPU burst of a process, rather than its total length. We use the term SJF because most people and textbooks use this term to refer to this type of scheduling.

As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

Using SJF scheduling, we would schedule these processes according to the following Gantt chart:



The waiting time is 3 milliseconds for process P_1, 16 milliseconds for process P_2, 9 milliseconds for process P_3, and 0 milliseconds for process P_4. Thus, the average waiting time is $(3 + 16 + 9 + 0)/4 = 7$ milliseconds. By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

The SJF scheduling algorithm is provably *optimal*, in that it gives the minimum average waiting time for a given set of processes. Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the *average* waiting time decreases.

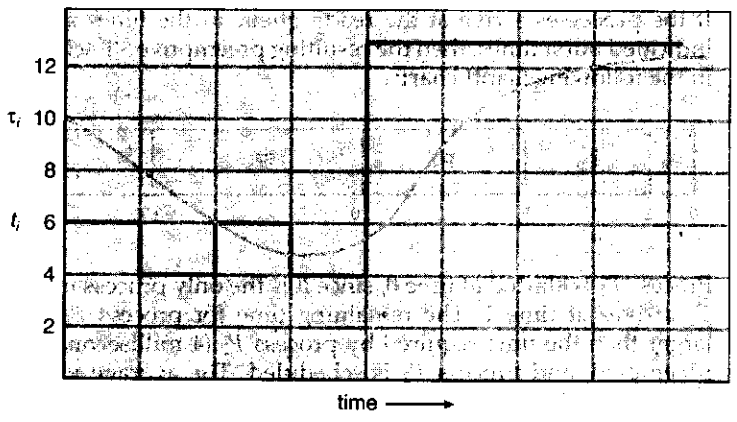
The real difficulty with the SJF algorithm is knowing the length of the next CPU request. For long-term (job) scheduling in a batch system, we can use as the length the process time limit that a user specifies when he submits the job. Thus, users are motivated to estimate the process time limit accurately, since a lower value may mean faster response. (Too low a value will cause a time-limit-exceeded error and require resubmission.) SJF scheduling is used frequently in long-term scheduling.

Although the SJF algorithm is optimal, it cannot be implemented at the level of short-term CPU scheduling. There is no way to know the length of the next CPU burst. One approach is to try to approximate SJF scheduling. We may not *know* the length of the next CPU burst, but we may be able to *predict* its value. We expect that the next CPU burst will be similar in length to the previous ones. Thus, by computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst.

The next CPU burst is generally predicted as an exponential average of the measured lengths of previous CPU bursts. Let t_n be the length of the nth CPU burst, and let τ_{n+1} be our predicted value for the next CPU burst. Then, for α, $0 \leq \alpha \leq 1$, define

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n.$$

This formula defines an **exponential average**. The value of t_n contains our most recent information; τ_n stores the past history. The parameter α controls the relative weight of recent and past history in our prediction. If $\alpha = 0$, then $\tau_{n+1} = \tau_n$, and recent history has no effect (current conditions are assumed to be transient); if $\alpha = 1$, then $\tau_{n+1} = t_n$, and only the most recent CPU burst



CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Figure 5.3 Prediction of the length of the next CPU burst.

matters (history is assumed to be old and irrelevant). More commonly, $\alpha = 1/2$, so recent history and past history are equally weighted. The initial τ_0 can be defined as a constant or as an overall system average. Figure 5.3 shows an exponential average with $\alpha = 1/2$ and $\tau_0 = 10$.

To understand the behavior of the exponential average, we can expand the formula for τ_{n+1} by substituting for τ_n, to find

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \cdots + (1-\alpha)^j \alpha t_{n-j} + \cdots + (1-\alpha)^{n+1}\tau_0.$$

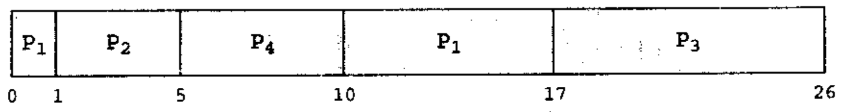
Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor.

The SJF algorithm can be either preemptive or nonpreemptive. The choice arises when a new process arrives at the ready queue while a previous process is still executing. The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process. A preemptive SJF algorithm will preempt the currently executing process, whereas a nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is sometimes called **shortest-remaining-time-first scheduling**.

As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:

Process	Arrival Time	Burst Time
P_1	0	8 (*)
P_2	1	4
P_3	2	9
P_4	3	5

If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt chart:



Process P_1 is started at time 0, since it is the only process in the queue. Process P_2 arrives at time 1. The remaining time for process P_1 (7 milliseconds) is larger than the time required by process P_2 (4 milliseconds), so process P_1 is preempted, and process P_2 is scheduled. The average waiting time for this example is $((10 - 1) + (1 - 1) + (17 - 2) + (5 - 3))/4 = 26/4 = 6.5$ milliseconds. Nonpreemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

5.3.3 Priority Scheduling

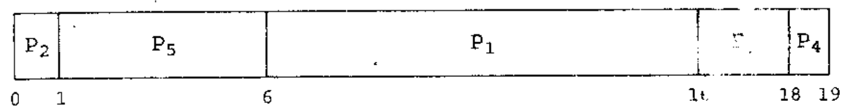
The SJF algorithm is a special case of the general **priority scheduling algorithm**. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.

Note that we discuss scheduling in terms of *high* priority and *low* priority. Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. However, there is no general agreement on whether 0 is the highest or lowest priority. Some systems use low numbers to represent low priority; others use low numbers for high priority. This difference can lead to confusion. In this text, we assume that low numbers represent high priority.

As an example, consider the following set of processes, assumed to have arrived at time 0, in the order P_1, P_2, $\cdots$, P_5, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Using priority scheduling, we would schedule these processes according to the following Gantt chart:



The average waiting time is 8.2 milliseconds.

Priorities can be defined either internally or externally. Internally defined priorities use some measurable quantity or quantities to compute the priority of a process. For example, time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities. External priorities are set by criteria outside the operating system, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political, factors.

Priority scheduling can be either preemptive or nonpreemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

A major problem with priority scheduling algorithms is **indefinite blocking**, or **starvation**. A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low-priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU. Generally, one of two things will happen. Either the process will eventually be run (at 2 A.M. Sunday, when the system is finally lightly loaded), or the computer system will eventually crash and lose all unfinished low-priority processes. (Rumor has it that, when they shut down the IBM 7094 at MIT in 1973, they found a low-priority process that had been submitted in 1967 and had not yet been run.)

A solution to the problem of indefinite blockage of low-priority processes is **aging**. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time. For example, if priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes. Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed. In fact, it would take no more than 32 hours for a priority-127 process to age to a priority-0 process.

5.3.4 Round-Robin Scheduling

The **round-robin (RR) scheduling algorithm** is designed especially for time-sharing systems. It is similar to FCFS scheduling, but preemption is added to switch between processes. A small unit of time, called a **time quantum** or time slice, is defined. A time quantum is generally from 10 to 100 milliseconds. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

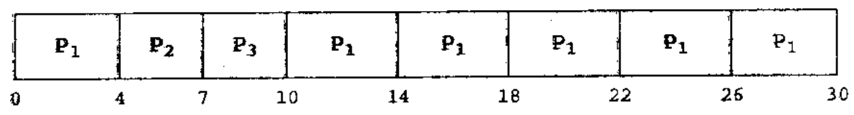
To implement RR scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

If we use a time quantum of 4 milliseconds, then process P_1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P_2. Since process P_2 does not need 4 milliseconds, it quits before its time quantum expires. The CPU is then given to the next process, process P_3. Once each process has received 1 time quantum, the CPU is returned to process P_1 for an additional time quantum. The resulting RR schedule is



The average waiting time is 17/3 = 5.66 milliseconds.

In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process). If a process's CPU burst exceeds 1 time quantum, that process is *preempted* and is put back in the ready queue. The RR scheduling algorithm is thus preemptive.

If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units. Each process must wait no longer than $(n - 1) \times q$ time units until its next time quantum. For example, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.

The performance of the RR algorithm depends heavily on the size of the time quantum. At one extreme, if the time quantum is extremely large, the RR policy is the same as the FCFS policy. If the time quantum is extremely small (say, 1 millisecond), the RR approach is called **processor sharing** and (in theory) creates the appearance that each of n processes has its own processor running at $1/n$ the speed of the real processor. This approach was used in Control Data Corporation (CDC) hardware to implement ten peripheral processors with only one set of hardware and ten sets of registers. The hardware executes one instruction for one set of registers, then goes on to the next. This cycle continues,

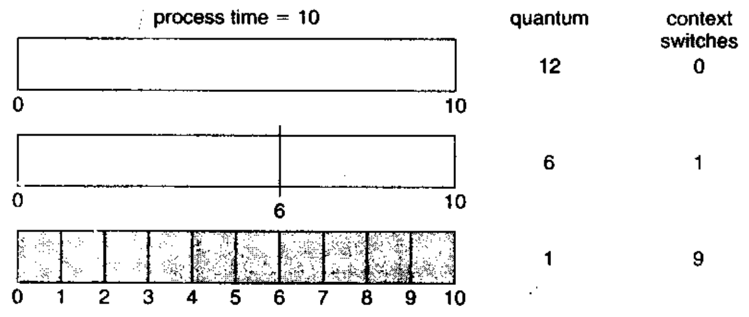


Figure 5.4 The way in which a smaller time quantum increases context switches.

resulting in ten slow processors rather than one fast one. (Actually, since the processor was much faster than memory and each instruction referenced memory, the processors were not much slower than ten real processors would have been.)

In software, we need also to consider the effect of context switching on the performance of RR scheduling. Let us assume that we have only one process of 10 time units. If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead. If the quantum is 6 time units, however, the process requires 2 quanta, resulting in a context switch. If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process accordingly (Figure 5.4).

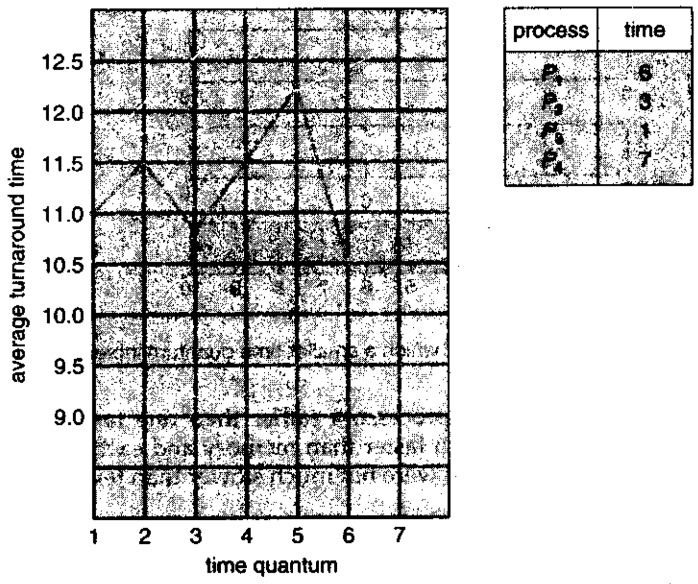
Thus, we want the time quantum to be large with respect to the context-switch time. If the context-switch time is approximately 10 percent of the time quantum, then about 10 percent of the CPU time will be spent in context switching. In practice, most modern systems have time quanta ranging from 10 to 100 milliseconds. The time required for a context switch is typically less than 10 microseconds; thus, the context-switch time is a small fraction of the time quantum.

Turnaround time also depends on the size of the time quantum. As we can see from Figure 5.5, the average turnaround time of a set of processes does not necessarily improve as the time-quantum size increases. In general, the average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum. For example, given three processes of 10 time units each and a quantum of 1 time unit, the average turnaround time is 29. If the time quantum is 10, however, the average turnaround time drops to 20. If context-switch time is added in, the average turnaround time increases for a smaller time quantum, since more context switches are required.

Although the time quantum should be large compared with the context-switch time, it should not be too large. If the time quantum is too large, RR scheduling degenerates to FCFS policy. A rule of thumb is that 80 percent of the CPU bursts should be shorter than the time quantum.

5.3.5 Multilevel Queue Scheduling

Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups. For example, a



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Figure 5.5 The way in which turnaround time varies with the time quantum.

common division is made between **foreground** (interactive) processes and **background** (batch) processes. These two types of processes have different response-time requirements and so may have different scheduling needs. In addition, foreground processes may have priority (externally defined) over background processes.

A **multilevel queue scheduling algorithm** partitions the ready queue into several separate queues (Figure 5.6). The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm. For example, separate queues might be used for foreground and background processes. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.

In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. For example, the foreground queue may have absolute priority over the background queue.

Let's look at an example of a multilevel queue scheduling algorithm with five queues, listed below in order of priority:

1. System processes
2. Interactive processes
3. Interactive editing processes
4. Batch processes
5. Student processes

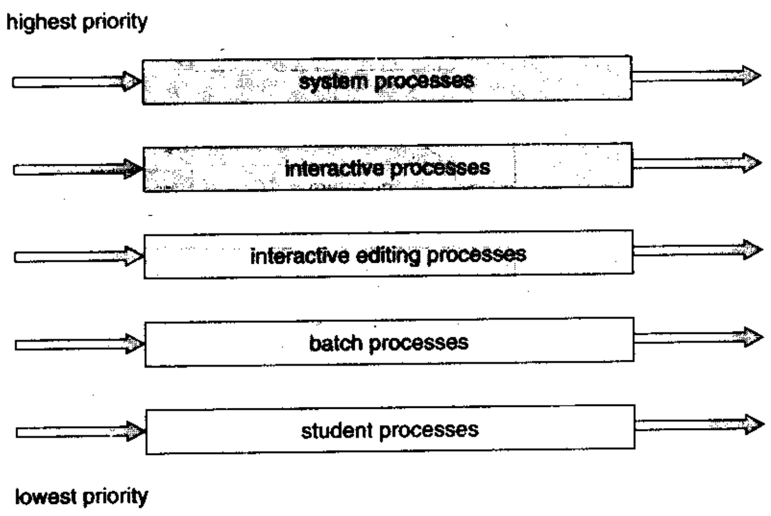


Figure 5.6 Multilevel queue scheduling.

Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.

Another possibility is to time-slice among the queues. Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes. For instance, in the foreground–background queue example, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes, whereas the background queue receives 20 percent of the CPU to give to its processes on an FCFS basis.

5.3.6 Multilevel Feedback-Queue Scheduling

Normally, when the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system. If there are separate queues for foreground and background processes, for example, processes do not move from one queue to the other, since processes do not change their foreground or background nature. This setup has the advantage of low scheduling overhead, but it is inflexible.

The **multilevel feedback-queue scheduling algorithm**, in contrast, allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues. In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

For example, consider a multilevel feedback-queue scheduler with three queues, numbered from 0 to 2 (Figure 5.7). The scheduler first executes all

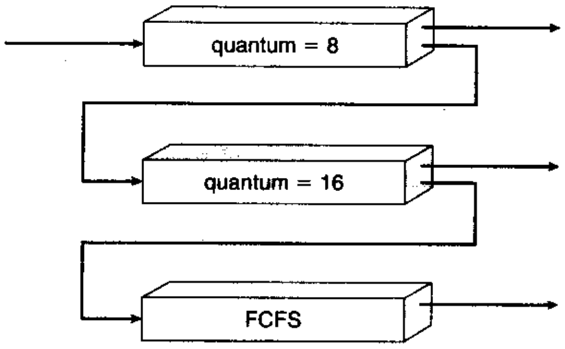


Figure 5.7 Multilevel feedback queues.

processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes in queue 2 will only be executed if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process in queue 1 will in turn be preempted by a process arriving for queue 0.

A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty.

This scheduling algorithm gives highest priority to any process with a CPU burst of 8 milliseconds or less. Such a process will quickly get the CPU, finish its CPU burst, and go off to its next I/O burst. Processes that need more than 8 but less than 24 milliseconds are also served quickly, although with lower priority than shorter processes. Long processes automatically sink to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.

In general, a multilevel feedback-queue scheduler is defined by the following parameters:

- The number of queues
- The scheduling algorithm for each queue
- The method used to determine when to upgrade a process to a higher-priority queue
- The method used to determine when to demote a process to a lower-priority queue
- The method used to determine which queue a process will enter when that process needs service

The definition of a multilevel feedback-queue scheduler makes it the most general CPU-scheduling algorithm. It can be configured to match a specific system under design. Unfortunately, it is also the most complex algorithm, since defining the best scheduler requires some means by which to select values for all the parameters.

5.4 Multiple-Processor Scheduling

Our discussion thus far has focused on the problems of scheduling the CPU in a system with a single processor. If multiple CPUs are available, **load sharing** becomes possible; however, the scheduling problem becomes correspondingly more complex. Many possibilities have been tried; and as we saw with single-processor CPU scheduling, there is no one best solution. Here, we discuss several concerns in multiprocessor scheduling. We concentrate on systems in which the processors are identical—**homogeneous**—in terms of their functionality; we can then use any available processor to run any process in the queue. (Note, however, that even with homogeneous multiprocessors, there are sometimes limitations on scheduling. Consider a system with an I/O device attached to a private bus of one processor. Processes that wish to use that device must be scheduled to run on that processor.)

5.4.1 Approaches to Multiple-Processor Scheduling

One approach to CPU scheduling in a multiprocessor system has all scheduling decisions, I/O processing, and other system activities handled by a single processor—the master server. The other processors execute only user code. This **asymmetric multiprocessing** is simple because only one processor accesses the system data structures, reducing the need for data sharing.

A second approach uses **symmetric multiprocessing (SMP)**, where each processor is self-scheduling. All processes may be in a common ready queue, or each processor may have its own private queue of ready processes. Regardless, scheduling proceeds by having the scheduler for each processor examine the ready queue and select a process to execute. As we shall see in Chapter 6, if we have multiple processors trying to access and update a common data structure, the scheduler must be programmed carefully: We must ensure that two processors do not choose the same process and that processes are not lost from the queue. Virtually all modern operating systems support SMP, including Windows XP, Windows 2000, Solaris, Linux, and Mac OS X.

In the remainder of this section, we will discuss issues concerning SMP systems.

5.4.2 Processor Affinity

Consider what happens to cache memory when a process has been running on a specific processor: The data most recently accessed by the process populates the cache for the processor; and as a result, successive memory accesses by the process are often satisfied in cache memory. Now consider what happens if the process migrates to another processor: The contents of cache memory must be invalidated for the processor being migrated from, and the cache for the processor being migrated to must be re-populated. Because of the high cost of invalidating and re-populating caches, most SMP systems try to avoid migration of processes from one processor to another and instead attempt to keep a process running on the same processor. This is known as **processor affinity**, meaning that a process has an affinity for the processor on which it is currently running.

Processor affinity takes several forms. When an operating system has a policy of attempting to keep a process running on the same processor—but not guaranteeing that it will do so— we have a situation known as **soft affinity**. Here, it is possible for a process to migrate between processors. Some systems —such as Linux—also provide system calls that support **hard affinity**, thereby allowing a process to specify that it is not to migrate to other processors.

5.4.3 Load Balancing

On SMP systems, it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor. Otherwise, one or more processors may sit idle while other processors have high workloads along with lists of processes awaiting the CPU. **Load balancing** attempts to keep the workload evenly distributed across all processors in an SMP system. It is important to note that load balancing is typically only necessary on systems where each processor has its own private queue of eligible processes to execute. On systems with a common run queue, load balancing is often unnecessary, because once a processor becomes idle, it immediately extracts a runnable process from the common run queue. It is also important to note, however, that in most contemporary operating systems supporting SMP, each processor does have a private queue of eligible processes.

There are two general approaches to load balancing: **push migration** and **pull migration**. With push migration, a specific task periodically checks the load on each processor and—if it finds an imbalance—evenly distributes the load by moving (or pushing) processes from overloaded to idle or less-busy processors. Pull migration occurs when an idle processor pulls a waiting task from a busy processor. Push and pull migration need not be mutually exclusive and are in fact often implemented in parallel on load-balancing systems. For example, the Linux scheduler (described in Section 5.6.3) and the ULE scheduler available for FreeBSD systems implement both techniques. Linux runs its load-balancing algorithm every 200 milliseconds (push migration) or whenever the run queue for a processor is empty (pull migration).

Interestingly, load balancing often counteracts the benefits of processor affinity, discussed in Section 5.4.2. That is, the benefit of keeping a process running on the same processor is that the process can take advantage of its data being in that processor's cache memory. By either pulling or pushing a process from one processor to another, we invalidate this benefit. As is often the case in systems engineering, there is no absolute rule concerning what policy is best. Thus, in some systems, an idle processor always pulls a process from a non-idle processor; and in other systems, processes are moved only if the imbalance exceeds a certain threshold.

5.4.4 Symmetric Multithreading

SMP systems allow several threads to run concurrently by providing multiple physical processors. An alternative strategy is to provide multiple *logical*—rather than *physical*—processors. Such a strategy is known as symmetric

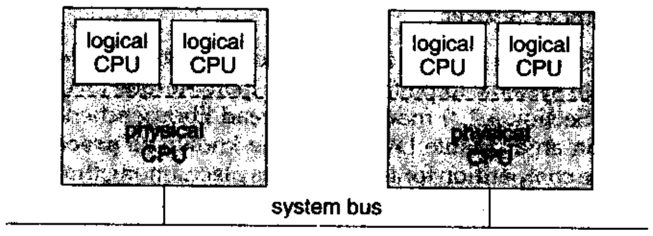


Figure 5.8 A typical SMT architecture

multithreading (or SMT); it has also been termed **hyperthreading technology** on Intel processors.

The idea behind SMT is to create multiple logical processors on the same physical processor, presenting a view of several logical processors to the operating system, even on a system with only a single physical processor. Each logical processor has its own **architecture state**, which includes general-purpose and machine-state registers. Furthermore, each logical processor is responsible for its own interrupt handling, meaning that interrupts are delivered to—and handled by—logical processors rather than physical ones. Otherwise, each logical processor shares the resources of its physical processor, such as cache memory and buses. Figure 5.8 illustrates a typical SMT architecture with two physical processors, each housing two logical processors. From the operating system's perspective, four processors are available for work on this system.

It is important to recognize that SMT is a feature provided in hardware, not software. That is, hardware must provide the representation of the architecture state for each logical processor, as well as interrupt handling. Operating systems need not necessarily be designed differently if they are to run on an SMT system; however, certain performance gains are possible if the operating system is aware that it is running on such a system. For example, consider a system with two physical processors, both of which are idle. The scheduler should first try scheduling separate threads on each physical processor rather than on separate logical processors on the same physical processor. Otherwise, both logical processors on one physical processor could be busy while the other physical processor remained idle.

5 Thread Scheduling

In Chapter 4, we introduced threads to the process model, distinguishing between *user-level* and *kernel-level* threads. On operating systems that support them, it is kernel-level threads—not processes—that are being scheduled by the operating system. User-level threads are managed by a thread library, and the kernel is unaware of them. To run on a CPU, user-level threads must ultimately be mapped to an associated kernel-level thread, although this mapping may be indirect and may use a lightweight process (LWP). In this section, we explore scheduling issues involving user-level and kernel-level threads and offer specific examples of scheduling for Pthreads.

5.5.1 Contention Scope

One distinction between user-level and kernel-level threads lies in how they are scheduled. On systems implementing the many-to-one (Section 4.2.1) and many-to-many (Section 4.2.3) models, the thread library schedules user-level threads to run on an available LWP, a scheme known as **process-contention scope (PCS)**, since competition for the CPU takes place among threads belonging to the same process. When we say the thread library *schedules* user threads onto available LWPs, we do not mean that the thread is actually running on a CPU; this would require the operating system to schedule the kernel thread onto a physical CPU. To decide which kernel thread to schedule onto a CPU, the kernel uses **system-contention scope (SCS)**. Competition for the CPU with SCS scheduling takes place among all threads in the system. Systems using the one-to-one model (such as Windows XP, Solaris 9, and Linux) schedule threads using only SCS.

Typically, PCS is done according to priority—the scheduler selects the runnable thread with the highest priority to run. User-level thread priorities are set by the programmer and are not adjusted by the thread library, although some thread libraries may allow the programmer to change the priority of a thread. It is important to note that PCS will typically preempt the thread currently running in favor of a higher-priority thread; however, there is no guarantee of time slicing (Section 5.3.4) among threads of equal priority.

5.5.2 Pthread Scheduling

We provided a sample POSIX Pthread program in Section 4.3.1, along with an introduction to thread creation with Pthreads. Now, we highlight the POSIX Pthread API that allows specifying either PCS or SCS during thread creation. Pthreads identifies the following contention scope values:

- PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling.
- PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling.

On systems implementing the many-to-many model (Section 4.2.3), the PTHREAD_SCOPE_PROCESS policy schedules user-level threads onto available LWPs. The number of LWPs is maintained by the thread library, perhaps using scheduler activations (Section 4.4.6). The PTHREAD_SCOPE_SYSTEM scheduling policy will create and bind an LWP for each user-level thread on many-to-many systems, effectively mapping threads using the one-to-one policy (Section 4.2.2).

The Pthread IPC provides the following two functions for getting—and setting—the contention scope policy:

- `pthread_attr_setscope(pthread_attr_t *attr, int scope)`
- `pthread_attr_getscope(pthread_attr_t *attr, int *scope)`

The first parameter for both functions contains a pointer to the attribute set for the thread. The second parameter for the `pthread_attr_setscope()` function is passed either the PTHREAD_SCOPE_SYSTEM or PTHREAD_SCOPE_PROCESS

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[])
{
   int i, scope;
   pthread_t tid[NUM_THREADS];
   pthread_attr_t attr;

   /* get the default attributes */
   pthread_attr_init(&attr);

   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
       printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
       printf("PTHREAD_SCOPE_SYSTEM");
      else
       fprintf(stderr, "Illegal scope value.\n");
   }

   /* set the scheduling algorithm to PCS or SCS */
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
       pthread_create(&tid[i],&attr,runner,NULL);

   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
       pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */

   pthread_exit(0);
}
```

Figure 5.9 Pthread scheduling API.

value, indicating how the contention scope is to be set. In the case of `pthread_attr_getscope()`, this second parameter contains a pointer to an `int` value that is set to the current value of the contention scope. If an error occurs, each of these functions returns non-zero values.

In Figure 5.9, we illustrate a Pthread program that first determines the existing contention scope and sets it to PTHREAD_SCOPE_PROCESS. It then creates five separate threads that will run using the SCS scheduling policy. Note that on some systems, only certain contention scope values are allowed. For example, Linux and Mac OS X systems allow only PTHREAD_SCOPE_SYSTEM.

5.6 Operating System Examples

We turn next to a description of the scheduling policies of the Solaris, Windows XP, and Linux operating systems. It is important to remember that we are describing the scheduling of kernel threads with Solaris and Linux. Recall that Linux does not distinguish between processes and threads; thus, we use the term *task* when discussing the Linux scheduler.

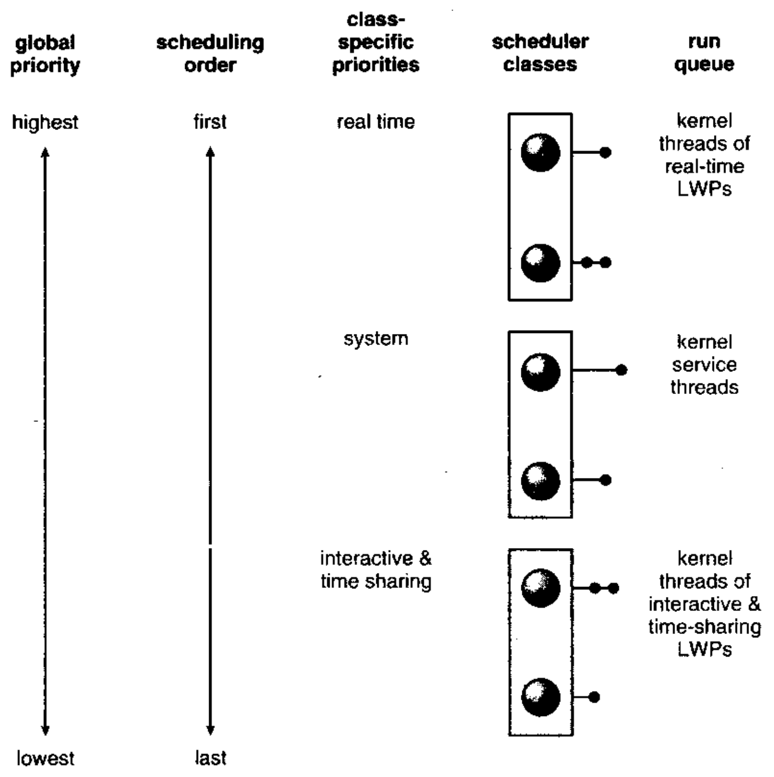

global priority
scheduling order
class-specific priorities
scheduler classes
run queue
highest
first
real time
kernel threads of real-time LWPs
system
kernel service threads
interactive & time sharing
kernel threads of interactive & time-sharing LWPs
lowest
last

Figure 5.10 Solaris scheduling.

5.6.1 Example: Solaris Scheduling

Solaris uses priority-based thread scheduling. It has defined four classes of scheduling, which are, in order of priority:

1. Real time
2. System
3. Time sharing
4. Interactive

Within each class there are different priorities and different scheduling algorithms. Solaris scheduling is illustrated in Figure 5.10.

The default scheduling class for a process is time sharing. The scheduling policy for time sharing dynamically alters priorities and assigns time slices of different lengths using a multilevel feedback queue. By default, there is an inverse relationship between priorities and time slices: The higher the priority, the smaller the time slice; and the lower the priority, the larger the time slice. Interactive processes typically have a higher priority; CPU-bound processes, a lower priority. This scheduling policy gives good response time for interactive processes and good throughput for CPU-bound processes. The interactive class uses the same scheduling policy as the time-sharing class, but it gives windowing applications a higher priority for better performance.

Figure 5.11 shows the dispatch table for scheduling interactive and time-sharing threads. These two scheduling classes include 60 priority levels, but

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Figure 5.11 Solaris dispatch table for interactive and time-sharing threads.

for brevity, we display only a handful. The dispatch table shown in Figure 5.11 contains the following fields:

- **Priority**. The class-dependent priority for the time-sharing and interactive classes. A higher number indicates a higher priority.
- **Time quantum**. The time quantum for the associated priority. This illustrates the inverse relationship between priorities and time quanta: The lowest priority (priority 0) has the highest time quantum (200 milliseconds), and the highest priority (priority 59) has the lowest time quantum (20 milliseconds).
- **Time quantum expired**. The new priority of a thread that has used its entire time quantum without blocking. Such threads are considered CPU-intensive. As shown in the table, these threads have their priorities lowered.
- **Return from sleep**. The priority of a thread that is returning from sleeping (such as waiting for I/O). As the table illustrates, when I/O is available for a waiting thread, its priority is boosted to between 50 and 59, thus supporting the scheduling policy of providing good response time for interactive processes.

Solaris 9 introduced two new scheduling classes: **fixed priority** and **fair share**. Threads in the fixed-priority class have the same priority range as those in the time-sharing class; however, their priorities are not dynamically adjusted. The fair-share scheduling class uses CPU **shares** instead of priorities to make scheduling decisions. CPU shares indicate entitlement to available CPU resources and are allocated to a set of processes (known as a **project**).

Solaris uses the system class to run kernel processes, such as the scheduler and paging daemon. Once established, the priority of a system process does not change. The system class is reserved for kernel use (user processes running in kernel mode are not in the systems class).

Threads in the real-time class are given the highest priority. This assignment allows a real-time process to have a guaranteed response from the system within a bounded period of time. A real-time process will run before a process in any other class. In general, however, few processes belong to the real-time class.

Each scheduling class includes a set of priorities. However, the scheduler converts the class-specific priorities into global priorities and selects the thread with the highest global priority to run. The selected thread runs on the CPU until it (1) blocks, (2) uses its time slice, or (3) is preempted by a higher-priority thread. If there are multiple threads with the same priority, the scheduler uses a round-robin queue. As mentioned, Solaris has traditionally used the many-to-many model (4.2.3) but with Solaris 9 switched to the one-to-one model (4.2.2).

5.6.2 Example: Windows XP Scheduling

Windows XP schedules threads using a priority-based, preemptive scheduling algorithm. The Windows XP scheduler ensures that the highest-priority thread will always run. The portion of the Windows XP kernel that handles scheduling is called the *dispatcher*. A thread selected to run by the dispatcher will run until it is preempted by a higher-priority thread, until it terminates, until its time quantum ends, or until it calls a blocking system call, such as for I/O. If a higher-priority real-time thread becomes ready while a lower-priority thread is running, the lower-priority thread will be preempted. This preemption gives a real-time thread preferential access to the CPU when the thread needs such access.

The dispatcher uses a 32-level priority scheme to determine the order of thread execution. Priorities are divided into two classes. The **variable class** contains threads having priorities from 1 to 15, and the **real-time class** contains threads with priorities ranging from 16 to 31. (There is also a thread running at priority 0 that is used for memory management.) The dispatcher uses a queue for each scheduling priority and traverses the set of queues from highest to lowest until it finds a thread that is ready to run. If no ready thread is found, the dispatcher will execute a special thread called the **idle thread**.

There is a relationship between the numeric priorities of the Windows XP kernel and the Win32 API. The Win32 API identifies several priority classes to which a process can belong. These include:

REALTIME_PRIORITY_CLASS

HIGH_PRIORITY_CLASS

ABOVE_NORMAL_PRIORITY_CLASS

NORMAL_PRIORITY_CLASS

BELOW_NORMAL_PRIORITY_CLASS

IDLE_PRIORITY_CLASS

Priorities in all classes except the REALTIME_PRIORITY_CLASS are variable, meaning that the priority of a thread belonging to one of these classes can change.

Within each of the priority classes is a relative priority. The values for relative priority include:

TIME_CRITICAL

HIGHEST

ABOVE_NORMAL

NORMAL

BELOW_NORMAL

LOWEST

IDLE

The priority of each thread is based on the priority class it belongs to and its relative priority within that class. This relationship is shown in Figure 5.12. The values of the priority classes appear in the top row. The left column contains the values for the relative priorities. For example, if the relative priority of a thread

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Figure 5.12 Windows XP priorities.

in the ABOVE_NORMAL_PRIORITY_CLASS is NORMAL, the numeric priority of that thread is 10.

Furthermore, each thread has a base priority representing a value in the priority range for the class the thread belongs to. By default, the base priority is the value of the NORMAL relative priority for that specific class. The base priorities for each priority class are:

REALTIME_PRIORITY_CLASS—24

HIGH_PRIORITY_CLASS—13

ABOVE_NORMAL_PRIORITY_CLASS—10

NORMAL_PRIORITY_CLASS—8

BELOW_NORMAL_PRIORITY_CLASS—6

IDLE_PRIORITY_CLASS—4

Processes are typically members of the NORMAL_PRIORITY_CLASS. A process will belong to this class unless the parent of the process was of the IDLE_PRIORITY_CLASS or unless another class was specified when the process was created. The initial priority of a thread is typically the base priority of the process the thread belongs to.

When a thread's time quantum runs out, that thread is interrupted; if the thread is in the variable-priority class, its priority is lowered. The priority is never lowered below the base priority, however. Lowering the thread's priority tends to limit the CPU consumption of compute-bound threads. When a variable-priority thread is released from a wait operation, the dispatcher boosts the priority. The amount of the boost depends on what the thread was waiting for; for example, a thread that was waiting for keyboard I/O would get a large increase, whereas a thread waiting for a disk operation would get a moderate one. This strategy tends to give good response times to interactive threads that are using the mouse and windows. It also enables I/O-bound threads to keep the I/O devices busy while permitting compute-bound threads to use spare CPU cycles in the background. This strategy is used by several time-sharing operating systems, including UNIX. In addition, the window with which the user is currently interacting receives a priority boost to enhance its response time.

When a user is running an interactive program, the system needs to provide especially good performance for that process. For this reason, Windows XP has a special scheduling rule for processes in the NORMAL_PRIORITY_CLASS. Windows XP distinguishes between the *foreground process* that is currently selected on the screen and the *background processes* that are not currently selected. When a process moves into the foreground, Windows XP increases the scheduling quantum by some factor—typically by 3. This increase gives the foreground process three times longer to run before a time-sharing preemption occurs.

5.6.3 Example: Linux Scheduling

Prior to version 2.5, the Linux kernel ran a variation of the traditional UNIX scheduling algorithm. Two problems with the traditional UNIX scheduler are that it does not provide adequate support for SMP systems and that it does not scale well as the number of tasks on the system grows. With version 2.5, the scheduler was overhauled, and the kernel now provides a scheduling algorithm that runs in constant time—known as $O(1)$—regardless of the number of tasks on the system. The new scheduler also provides increased support for SMP, including processor affinity and load balancing, as well as providing fairness and support for interactive tasks.

The Linux scheduler is a preemptive, priority-based algorithm with two separate priority ranges: a **real-time** range from 0 to 99 and a **nice** value ranging from 100 to 140. These two ranges map into a global priority scheme whereby numerically lower values indicate higher priorities.

Unlike schedulers for many other systems, including Solaris (5.6.1) and Windows XP (5.6.2), Linux assigns higher-priority tasks longer time quanta and lower-priority tasks shorter time quanta. The relationship between priorities and time-slice length is shown in Figure 5.13.

A runnable task is considered eligible for execution on the CPU as long as it has time remaining in its time slice. When a task has exhausted its time slice, it is considered expired and is not eligible for execution again until all other tasks have also exhausted their time quanta. The kernel maintains a list

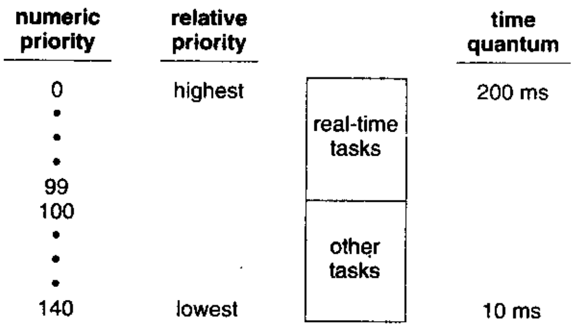


Figure 5.13 The relationship between priorities and time-slice length.

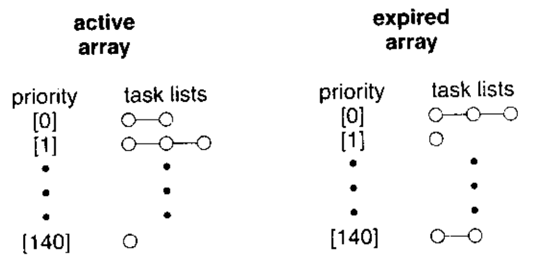


Figure 5.14 List of tasks indexed according to priority.

of all runnable tasks in a **runqueue** data structure. Because of its support for SMP, each processor maintains its own runqueue and schedules itself independently. Each runqueue contains two priority arrays—**active** and **expired**. The active array contains all tasks with time remaining in their time slices, and the expired array contains all expired tasks. Each of these priority arrays contains a list of tasks indexed according to priority (Figure 5.14). The scheduler chooses the task with the highest priority from the active array for execution on the CPU. On multiprocessor machines, this means that each processor is scheduling the highest-priority task from its own runqueue structure. When all tasks have exhausted their time slices (that is, the active array is empty), the two priority arrays are exchanged; the expired array becomes the active array, and vice versa.

Linux implements real-time scheduling as defined by POSIX.1b, which is fully described in Section 5.5.2. Real-time tasks are assigned static priorities. All other tasks have dynamic priorities that are based on their *nice* values plus or minus the value 5. The interactivity of a task determines whether the value 5 will be added to or subtracted from the *nice* value. A task's interactivity is determined by how long it has been sleeping while waiting for I/O. Tasks that are more interactive typically have longer sleep times and therefore are more likely to have adjustments closer to -5, as the scheduler favors interactive tasks. The result of such adjustments will be higher priorities for these tasks. Conversely, tasks with shorter sleep times are often more CPU-bound and thus will have their priorities lowered.

The recalculation of a task's dynamic priority occurs when the task has exhausted its time quantum and is to be moved to the expired array. Thus, when the two arrays are exchanged, all tasks in the new active array have been assigned new priorities and corresponding time slices.

5.7

How do we select a CPU scheduling algorithm for a particular system? As we saw in Section 5.3, there are many scheduling algorithms, each with its own parameters. As a result, selecting an algorithm can be difficult.

The first problem is defining the criteria to be used in selecting an algorithm. As we saw in Section 5.2, criteria are often defined in terms of CPU utilization, response time, or throughput. To select an algorithm, we must first define the relative importance of these measures. Our criteria may include several measures, such as:

> Maximizing CPU utilization under the constraint that the maximum response time is 1 second
>
> Maximizing throughput such that turnaround time is (on average) linearly proportional to total execution time

Once the selection criteria have been defined, we want to evaluate the algorithms under consideration. We next describe the various evaluation methods we can use.

5.7.1 Deterministic Modeling

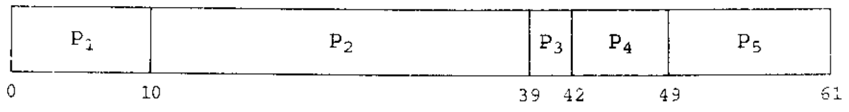
One major class of evaluation methods is **analytic evaluation**. Analytic evaluation uses the given algorithm and the system workload to produce a formula or number that evaluates the performance of the algorithm for that workload.

One type of analytic evaluation is **deterministic modeling**. This method takes a particular predetermined workload and defines the performance of each algorithm for that workload. For example, assume that we have the workload shown below. All five processes arrive at time 0, in the order given, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

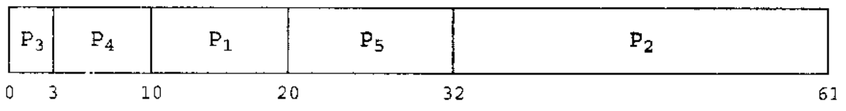
Consider the FCFS, SJF, and RR (quantum = 10 milliseconds) scheduling algorithms for this set of processes. Which algorithm would give the minimum average waiting time?

For the FCFS algorithm, we would execute the processes as



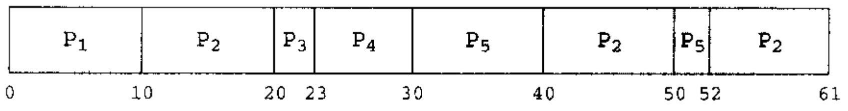
The waiting time is 0 milliseconds for process P_1, 10 milliseconds for process P_2, 39 milliseconds for process P_3, 42 milliseconds for process P_4, and 49 milliseconds for process P_5. Thus, the average waiting time is (0 + 10 + 39 + 42 + 49)/5 = 28 milliseconds.

With nonpreemptive SJF scheduling, we execute the processes as



The waiting time is 10 milliseconds for process P_1, 32 milliseconds for process P_2, 0 milliseconds for process P_3, 3 milliseconds for process P_4, and 20 milliseconds for process P_5. Thus, the average waiting time is (10 + 32 + 0 + 3 + 20)/5 = 13 milliseconds.

With the RR algorithm, we execute the processes as



The waiting time is 0 milliseconds for process P_1, 32 milliseconds for process P_2, 20 milliseconds for process P_3, 23 milliseconds for process P_4, and 40 milliseconds for process P_5. Thus, the average waiting time is (0 + 32 + 20 + 23 + 40)/5 = 23 milliseconds.

We see that, *in this case,* the average waiting time obtained with the SJF policy is less than half that obtained with FCFS scheduling; the RR algorithm gives us an intermediate value.

Deterministic modeling is simple and fast. It gives us exact numbers, allowing us to compare the algorithms. However, it requires exact numbers for input, and its answers apply only to those cases. The main uses of deterministic modeling are in describing scheduling algorithms and providing examples. In cases where we are running the same program over and over again and can measure the program's processing requirements exactly, we may be able to use deterministic modeling to select a scheduling algorithm. Furthermore, over a set of examples, deterministic modeling may indicate trends that can then be analyzed and proved separately. For example, it can be shown that, for the environment described (all processes and their times available at time 0), the SJF policy will always result in the minimum waiting time.

5.7.2 Queueing Models

On many systems, the processes that are run vary from day to day, so there is no static set of processes (or times) to use for deterministic modeling. What can be determined, however, is the distribution of CPU and I/O bursts. These distributions can be measured and then approximated or simply estimated. The result is a mathematical formula describing the probability of a particular CPU burst. Commonly, this distribution is exponential and is described by its mean. Similarly, we can describe the distribution of times when processes arrive in the system (the arrival-time distribution). From these two distributions, it is possible to compute the average throughput, utilization, waiting time, and so on for most algorithms.

The computer system is described as a network of servers. Each server has a queue of waiting processes. The CPU is a server with its ready queue, as is the I/O system with its device queues. Knowing arrival rates and service rates, we can compute utilization, average queue length, average wait time, and so on. This area of study is called **queueing-network analysis**.

As an example, let n be the average queue length (excluding the process being serviced), let W be the average waiting time in the queue, and let λ be the average arrival rate for new processes in the queue (such as three processes per second). We expect that during the time W that a process waits, $\lambda \times W$ new processes will arrive in the queue. If the system is in a steady state, then the number of processes leaving the queue must be equal to the number of processes that arrive. Thus,

$$n = \lambda \times W.$$

This equation, known as **Little's formula**, is particularly useful because it is valid for any scheduling algorithm and arrival distribution.

We can use Little's formula to compute one of the three variables, if we know the other two. For example, if we know that 7 processes arrive every second (on average), and that there are normally 14 processes in the queue, then we can compute the average waiting time per process as 2 seconds.

Queueing analysis can be useful in comparing scheduling algorithms, but it also has limitations. At the moment, the classes of algorithms and distributions that can be handled are fairly limited. The mathematics of complicated algorithms and distributions can be difficult to work with. Thus, arrival and service distributions are often defined in mathematically tractable —but unrealistic—ways. It is also generally necessary to make a number of independent assumptions, which may not be accurate. As a result of these difficulties, queueing models are often only approximations of real systems, and the accuracy of the computed results may be questionable.

5.7.3 Simulations

To get a more accurate evaluation of scheduling algorithms, we can use **simulations**. Running simulations involves programming a model of the computer system. Software data structures represent the major components of the system. The simulator has a variable representing a clock; as this variable's value is increased, the simulator modifies the system state to reflect the activities of the devices, the processes, and the scheduler. As the simulation executes, statistics that indicate algorithm performance are gathered and printed.

The data to drive the simulation can be generated in several ways. The most common method uses a random-number generator, which is programmed to generate processes, CPU burst times, arrivals, departures, and so on, according to probability distributions. The distributions can be defined mathematically (uniform, exponential, Poisson) or empirically. If a distribution is to be defined empirically, measurements of the actual system under study are taken. The results define the distribution of events in the real system; this distribution can then be used to drive the simulation.

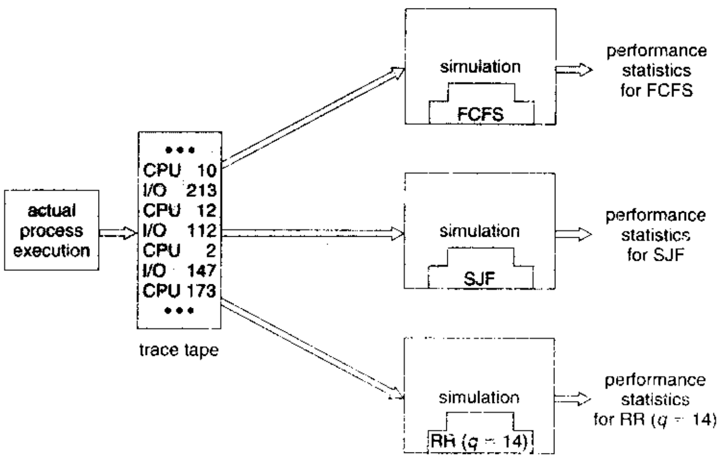


Figure 5.15 Evaluation of CPU schedulers by simulation.

A distribution-driven simulation may be inaccurate, however, because of relationships between successive events in the real system. The frequency distribution indicates only how many instances of each event occur; it does not indicate anything about the order of their occurrence. To correct this problem, we can use **trace tapes**. We create a trace tape by monitoring the real system and recording the sequence of actual events (Figure 5.15). We then use this sequence to drive the simulation. Trace tapes provide an excellent way to compare two algorithms on exactly the same set of real inputs. This method can produce accurate results for its inputs.

Simulations can be expensive, often requiring hours of computer time. A more detailed simulation provides more accurate results, but it also requires more computer time. In addition, trace tapes can require large amounts of storage space. Finally, the design, coding, and debugging of the simulator can be a major task.

5.7.4 Implementation

Even a simulation is of limited accuracy. The only completely accurate way to evaluate a scheduling algorithm is to code it up, put it in the operating system, and see how it works. This approach puts the actual algorithm in the real system for evaluation under real operating conditions.

The major difficulty with this approach is the high cost. The expense is incurred not only in coding the algorithm and modifying the operating system to support it (along with its required data structures) but also in the reaction of the users to a constantly changing operating system. Most users are not interested in building a better operating system; they merely want to get their processes executed and use their results. A constantly changing operating system does not help the users to get their work done.

Another difficulty is that the environment in which the algorithm is used will change. The environment will change not only in the usual way, as new programs are written and the types of problems change, but also as a result of the performance of the scheduler. If short processes are given priority, then users may break larger processes into sets of smaller processes. If interactive processes are given priority over noninteractive processes, then users may switch to interactive use.

For example, researchers designed one system that classified interactive and noninteractive processes automatically by looking at the amount of terminal I/O. If a process did not input or output to the terminal in a 1-second interval, the process was classified as noninteractive and was moved to a lower-priority queue. In response to this policy, one programmer modified his programs to write an arbitrary character to the terminal at regular intervals of less than 1 second. The system gave his programs a high priority, even though the terminal output was completely meaningless.

The most flexible scheduling algorithms are those that can be altered by the system managers or by the users so that they can be tuned for a specific application or set of applications. For instance, a workstation that performs high-end graphical applications may have scheduling needs different from those of a web server or file server. Some operating systems—particularly several versions of UNIX—allow the system manager to fine-tune the scheduling parameters for a particular system configuration. For example, Solaris provides the `dispadmin` command to allow the system administrator to modify the parameters of the scheduling classes described in Section 5.6.1.

Another approach is to use APIs that modify the priority of a process or thread. The Java, /POSIX, and /WinAPI/ provide such functions. The downfall of this approach is that performance tuning a system or application most often does not result in improved performance in more general situations.

5.8

CPU scheduling is the task of selecting a waiting process from the ready queue and allocating the CPU to it. The CPU is allocated to the selected process by the dispatcher.

First-come, first-served (FCFS) scheduling is the simplest scheduling algorithm, but it can cause short processes to wait for very long processes. Shortest-job-first (SJF) scheduling is provably optimal, providing the shortest average waiting time. Implementing SJF scheduling is difficult, however, because predicting the length of the next CPU burst is difficult. The SJF algorithm is a special case of the general priority scheduling algorithm, which simply allocates the CPU to the highest-priority process. Both priority and SJF scheduling may suffer from starvation. Aging is a technique to prevent starvation.

Round-robin (RR) scheduling is more appropriate for a time-shared (interactive) system. RR scheduling allocates the CPU to the first process in the ready queue for q time units, where q is the time quantum. After q time units, if the process has not relinquished the CPU, it is preempted, and the process is put at the tail of the ready queue. The major problem is the selection of the time quantum. If the quantum is too large, RR scheduling degenerates to FCFS scheduling; if the quantum is too small, scheduling overhead in the form of context-switch time becomes excessive.

The FCFS algorithm is nonpreemptive; the RR algorithm is preemptive. The SJF and priority algorithms may be either preemptive or nonpreemptive.

Multilevel queue algorithms allow different algorithms to be used for different classes of processes. The most common model includes a foreground interactive queue that uses RR scheduling and a background batch queue that uses FCFS scheduling. Multilevel feedback queues allow processes to move from one queue to another.

Many contemporary computer systems support multiple processors and allow each processor to schedule itself independently. Typically, each processor maintains its own private queue of processes (or threads), all of which are available to run. Issues related to multiprocessor scheduling include processor affinity and load balancing.

Operating systems supporting threads at the kernel level must schedule threads—not processes—for execution. This is the case with Solaris and Windows XP. Both of these systems schedule threads using preemptive, priority-based scheduling algorithms, including support for real-time threads. The Linux process scheduler uses a priority-based algorithm with real-time support as well. The scheduling algorithms for these three operating systems typically favor interactive over batch and CPU-bound processes.

The wide variety of scheduling algorithms demands that we have methods to select among algorithms. Analytic methods use mathematical analysis to determine the performance of an algorithm. Simulation methods determine performance by imitating the scheduling algorithm on a "representative" sample of processes and computing the resulting performance. However, simulation can at best provide an approximation of actual system performance; the only reliable technique for evaluating a scheduling algorithm is to implement the algorithm on an actual system and monitor its performance in a "real-world" environment.

5.1 Discuss how the following pairs of scheduling criteria conflict in certain settings.

a. CPU utilization and response time

b. Average turnaround time and maximum waiting time

c. I/O device utilization and CPU utilization

5.2 Consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

The processes are assumed to have arrived in the order P_1, P_2, P_3, P_4, P_5, all at time 0.

a. Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, nonpreemptive priority (a smaller priority number implies a higher priority), and RR (quantum = 1).

b. What is the turnaround time of each process for each of the scheduling algorithms in part a?

c. What is the waiting time of each process for each of the scheduling algorithms in part a?

d. Which of the algorithms in part a results in the minimum average waiting time (over all processes)?

5.3 Why is it important for the scheduler to distinguish I/O-bound programs from CPU-bound programs?

5.4 Which of the following scheduling algorithms could result in starvation?

a. First-come, first-served

b. Shortest job first

c. Round robin

d. Priority

5.5 Consider a system running ten I/O-bound tasks and one CPU-bound task. Assume that the I/O-bound tasks issue an I/O operation once for every millisecond of CPU computing and that each I/O operation takes 10 milliseconds to complete. Also assume that the context-switching overhead is 0.1 millisecond and that all processes are long-running tasks. What is the CPU utilization for a round-robin scheduler when:

a. The time quantum is 1 millisecond

b. The time quantum is 10 milliseconds

5.6 Consider a system implementing multilevel queue scheduling. What strategy can a computer user employ to maximize the amount of CPU time allocated to the user's process?

5.7 Explain the differences in the degree to which the following scheduling algorithms discriminate in favor of short processes:

a. FCFS

b. RR

c. Multilevel feedback queues

5.8 Using the Windows XP scheduling algorithm, what is the numeric priority of a thread for the following scenarios?

a. A thread in the REALTIME_PRIORITY_CLASS with a relative priority of HIGHEST

b. A thread in the NORMAL_PRIORITY_CLASS with a relative priority of NORMAL

c. A thread in the HIGH_PRIORITY_CLASS with a relative priority of ABOVE_NORMAL

5.9 The traditional UNIX scheduler enforces an inverse relationship between priority numbers and priorities: The higher the number, the lower the priority. The scheduler recalculates process priorities once per second using the following function:

Priority = (recent CPU usage / 2) + base

where base = 60 and *recent CPU usage* refers to a value indicating how often a process has used the CPU since priorities were last recalculated.

Assume that recent CPU usage for process P_1 is 40, process P_2 is 18, and process P_3 is 10. What will be the new priorities for these three processes when priorities are recalculated? Based on this information, does the traditional UNIX scheduler raise or lower the relative priority of a CPU-bound process?

Feedback queues were originally implemented on the CTSS system described in Corbato et al. [1962]. This feedback queue scheduling system was analyzed by Schrage [1967].

Anderson et al. [1989], Lewis and Berg [1998], and Philbin et al. [1996] talked about thread scheduling. Multiprocessor scheduling was discussed by Tucker and Gupta [1989], Zahorjan and McCann [1990], Feitelson and Rudolph [1990], Leutenegger and Vernon [1990], Blumofe and Leiserson [1994], Polychronopoulos and Kuck [1987], and Lucco [1992]. Scheduling techniques that take into account information regarding process execution times from previous runs were described in Fisher [1981], Hall et al. [1996], and Lowney et al. [1993].

Scheduling in real-time systems was discussed by Liu and Layland [1973], Abbot [1984], Jensen et al. [1985], Hong et al. [1989], and Khanna et al. [1992]. A special issue of *Operating System Review* on real-time operating systems was edited by Zhao [1989].

Fair-share schedulers were covered by Henry [1984], Woodside [1986], and Kay and Lauder [1988].

Scheduling policies used in the UNIX V operating system were described by Bach [1987]; those for UNIX BSD 4.4 were presented by McKusick et al. [1996]; and those for the Mach operating system were discussed by Black [1990]. Bovet and Cesati [2002] covered scheduling in Linux. Solaris scheduling was described by Mauro and McDougall [2001]. Solomon [1998] and Solomon and Russinovich [2000] discussed scheduling in Windows NT and Windows 2000, respectively. Butenhof [1997] and Lewis and Berg [1998] described scheduling in Pthreads systems.

Part Three

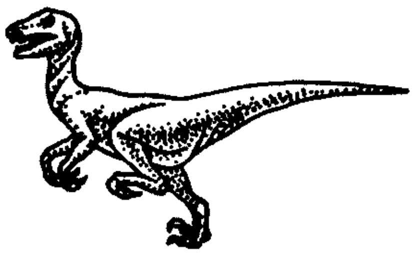
A *cooperating process* is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages. The former case is achieved through the use of lightweight processes or threads.

Concurrent access to shared data may result in data inconsistency. There are various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

Some of these mechanisms may result in a situation where processes are waiting indefinitely in a waiting state. This situation is called a *deadlock*. There are a number of different methods that an operating system can use to prevent or deal with deadlocks.

CHAPTER

[illegible]



A cooperating process is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages. The former case is achieved through the use of lightweight processes or threads, which we discussed in Chapter 4. Concurrent access to shared data may result in data inconsistency. In this chapter, we discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

6.1 [illegible]

In Chapter 3, we developed a model of a system consisting of cooperating sequential processes or threads, all running asynchronously and possibly sharing data. We illustrated this model with the producer–consumer problem, which is representative of operating systems. Specifically, in Section 3.4.1, we described how a bounded buffer could be used to enable processes to share memory.

Let us return to our consideration of the bounded buffer. As we pointed out, our solution allows at most BUFFER_SIZE − 1 items in the buffer at the same time. Suppose we want to modify the algorithm to remedy this deficiency. One possibility is to add an integer variable `counter`, initialized to 0. `counter` is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer. The code for the producer process can be modified as shown in Figure 6.1. The code for the consumer process can be modified as shown in Figure 6.2.

Although both the producer and consumer routines are correct separately, they may not function correctly when executed concurrently. As an illustration, suppose that the value of the variable `counter` is currently 5 and that the producer and consumer processes execute the statements "`counter++`" and "`counter--`" concurrently. Following the execution of these two statements, the value of the variable `counter` may be 4, 5, or 6! The only correct result,

```
while (true)
{
     /* produce an item in nextProduced */
     while (counter == BUFFER_SIZE)
        ; /* do nothing */
     buffer[in] = nextProduced;
     in = (in + 1) % BUFFER_SIZE;
     counter++;
}
```

Figure 6.1 The code for the producer process.

though, is `counter == 5`, which is generated correctly if the producer and consumer execute separately.

We can show that the value of `counter` may be incorrect as follows. Note that the statement "`counter++`" may be implemented in machine language (on a typical machine) as

$$\begin{aligned} register_1 &= \texttt{counter} \\ register_1 &= register_1 + 1 \\ \texttt{counter} &= register_1 \end{aligned}$$

where $register_1$ is a local CPU register. Similarly, the statement "`counter--`" is implemented as follows:

$$\begin{aligned} register_2 &= \texttt{counter} \\ register_2 &= register_2 - 1 \\ \texttt{counter} &= register_2 \end{aligned}$$

where again $register_2$ is a local CPU register. Even though $register_1$ and $register_2$ may be the same physical register (an accumulator, say), remember that the contents of this register will be saved and restored by the interrupt handler (Section 1.2.3).

The concurrent execution of "`counter++`" and "`counter--`" is equivalent to a sequential execution where the lower-level statements presented pre-

```
while (true)
{
     while (counter == 0)
        ; /* do nothing */
     nextConsumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     counter--;
     /* consume the item in nextConsumed */
}
```

Figure 6.2 The code for the consumer process.

viously are interleaved in some arbitrary order (but the order within each high-level statement is preserved). One such interleaving is

T_0:	*producer*	execute	$register_1$ = `counter`	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2$ = `counter`	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	`counter` = $register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	`counter` = $register_2$	$\{counter = 4\}$

Notice that we have arrived at the incorrect state "`counter` == 4", indicating that four buffers are full, when, in fact, five buffers are full. If we reversed the order of the statements at T_4 and T_5, we would arrive at the incorrect state "`counter` == 6".

We would arrive at this incorrect state because we allowed both processes to manipulate the variable `counter` concurrently. A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**. To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable `counter`. To make such a guarantee, we require that the processes be synchronized in some way.

Situations such as the one just described occur frequently in operating systems as different parts of the system manipulate resources. Clearly, we want the resulting changes not to interfere with one another. Because of the importance of this issue, a major portion of this chapter is concerned with **process synchronization** and **coordination**.

6.2 The Critical-Section Problem

Consider a system consisting of n processes $\{P_0, P_1, ..., P_{n-1}\}$. Each process has a segment of code, called a **critical section**, in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. The *critical-section problem* is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section**. The general structure of a typical process P_i is shown in Figure 6.3. The entry section and exit section are enclosed in boxes to highlight these important segments of code.

A solution to the critical-section problem must satisfy the following three requirements:

Mutual exclusion. If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

```
do {

     entry section

          critical section

     exit section

          remainder section

} while (TRUE);
```

Figure 6.3 General structure of a typical process P_i.

2. **Progress**. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.

3. **Bounded waiting**. There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

We assume that each process is executing at a nonzero speed. However, we can make no assumption concerning the **relative speed** of the n processes.

At a given point in time, many kernel-mode processes may be active in the operating system. As a result, the code implementing an operating system (*kernel code*) is subject to several possible race conditions. Consider as an example a kernel data structure that maintains a list of all open files in the system. This list must be modified when a new file is opened or closed (adding the file to the list or removing it from the list). If two processes were to open files simultaneously, the separate updates to this list could result in a race condition. Other kernel data structures that are prone to possible race conditions include structures for maintaining memory allocation, for maintaining process lists, and for interrupt handling. It is up to kernel developers to ensure that the operating system is free from such race conditions.

Two general approaches are used to handle critical sections in operating systems: (1) **preemptive kernels** and (2) **nonpreemptive kernels**. A preemptive kernel allows a process to be preempted while it is running in kernel mode. A nonpreemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU. Obviously, a nonpreemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time. We cannot say the same about nonpreemptive kernels, so they must be carefully designed to ensure that shared kernel data are free from race conditions. Preemptive kernels are especially difficult to design for SMP architectures, since in these environments it is possible for two kernel-mode processes to run simultaneously on different processors.

Why, then, would anyone favor a preemptive kernel over a nonpreemptive one? A preemptive kernel is more suitable for real-time programming, as it will allow a real-time process to preempt a process currently running in the kernel. Furthermore, a preemptive kernel may be more responsive, since there is less risk that a kernel-mode process will run for an arbitrarily long period before relinquishing the processor to waiting processes. Of course, this effect can be minimized by designing kernel code that does not behave in this way.

Windows XP and Windows 2000 are nonpreemptive kernels, as is the traditional UNIX kernel. Prior to Linux 2.6, the Linux kernel was nonpreemptive as well. However, with the release of the 2.6 kernel, Linux changed to the preemptive model. Several commercial versions of UNIX are preemptive, including Solaris and IRIX.

6.3 Peterson's Solution

Next, we illustrate a classic software-based solution to the critical-section problem known as **Peterson's solution**. Because of the way modern computer architectures perform basic machine-language instructions, such as `load` and `store`, there are no guarantees that Peterson's solution will work correctly on such architectures. However, we present the solution because it provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting requirements.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered P_0 and P_1. For convenience, when presenting P_i, we use P_j to denote the other process; that is, `j` equals `1 - i`.

Peterson's solution requires two data items to be shared between the two processes:

```
int turn;
boolean flag[2];
```

The variable `turn` indicates whose turn it is to enter its critical section. That is, if `turn == i`, then process P_i is allowed to execute in its critical section. The `flag` array is used to indicate if a process *is ready* to enter its critical section. For example, if `flag[i]` is `true`, this value indicates that P_i is ready to enter its critical section. With an explanation of these data structures complete, we are now ready to describe the algorithm shown in Figure 6.4.

To enter the critical section, process P_i first sets `flag[i]` to be `true` and then sets `turn` to the value `j`, thereby asserting that if the other process wishes to enter the critical section, it can do so. If both processes try to enter at the same time, `turn` will be set to both `i` and `j` at roughly the same time. Only one of these assignments will last; the other will occur but will be overwritten

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

}while (TRUE);
```

Figure 6.4 The structure of process P_i in Peterson's solution.

immediately. The eventual value of `turn` decides which of the two processes is allowed to enter its critical section first.

We now prove that this solution is correct. We need to show that: Rob

1. Mutual exclusion is preserved.
2. The progress requirement is satisfied.
3. The bounded-waiting requirement is met.

To prove property 1, we note that each P_i enters its critical section only if either `flag[j]` == `false` or `turn` == `i`. Also note that, if both processes can be executing in their critical sections at the same time, then `flag[0]` == `flag[1]` == `true`. These two observations imply that P_0 and P_1 could not have successfully executed their `while` statements at about the same time, since the value of `turn` can be either 0 or 1 but cannot be both. Hence, one of the processes —say P_j—must have successfully executed the while statement, whereas P_i had to execute at least one additional statement ("`turn` == `j`"). However, since, at that time, `flag[j]` == `true`, and `turn` == `j`, and this condition will persist as long as P_j is in its critical section, the result follows: Mutual exclusion is preserved.

To prove properties 2 and 3, we note that a process P_i can be prevented from entering the critical section only if it is stuck in the while loop with the condition `flag[j]` == `true` and `turn` == `j`; this loop is the only one possible. If P_j is not ready to enter the critical section, then `flag[j]` == `false`, and P_i can enter its critical section. If P_j has set `flag[j]` to `true` and is also executing in its `while` statement, then either `turn` == `i` or `turn` == `j`. If `turn` == `i`, then P_i will enter the critical section. If `turn` == `j`, then P_j will enter the critical section. However, once P_j exits its critical section, it will reset `flag[j]` to `false`, allowing P_i to enter its critical section. If P_j resets `flag[j]` to `true`, it must also set `turn` to `i`. Thus, since P_i does not change the value of the variable `turn` while executing the `while` statement, P_i will enter the critical section (progress) after at most one entry by P_j (bounded waiting).

```
do {

    acquire lock

        critical section

    release lock

        remainder section

} while (TRUE);
```

Figure 6.5 Solution to the critical-section problem using locks.

6.4 Synchronization Hardware

We have just described one software-based solution to the critical-section problem. In general, we can state that any solution to the critical-section problem requires a simple tool—a **lock**. Ragce conditions are prevented by requiring that critical regions be protected by locks. That is, a process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section. This is illustrated in Figure 6.5.

In the following discussions, we explore several more solutions to the critical-section problem using techniques ranging from hardware to software-based APIs available to application programmers. All these solutions are based on the premise of locking; however, as we shall see, the design of such locks can be quite sophisticated.

Hardware features can make any programming task easier and improve system efficiency. In this section, we present some simple hardware instructions that are available on many systems and show how they can be used effectively in solving the critical-section problem.

The critical-section problem could be solved simply in a uniprocessor environment if we could prevent interrupts from occurring while a shared variable was being modified. In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable. This is the approach taken by nonpreemptive kernels.

Unfortunately, this solution is not as feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming, as the

```
boolean TestAndSet(boolean *target) {
   boolean rv = *target;
   *target = TRUE;
   return rv;
}
```

Figure 6.6 The definition of the TestAndSet() instruction.

```
do {
   while (TestAndSetLock(&lock))
      ; // do nothing

      // critical section

   lock = FALSE;

      // remainder section
}while (TRUE);
```

Figure 6.7 Mutual-exclusion implementation with `TestAndSet()`.

message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases. Also, consider the effect on a system's clock, if the clock is kept updated by interrupts.

Many modern computer systems therefore provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words **atomically**—that is, as one uninterruptible unit. We can use these special instructions to solve the critical-section problem in a relatively simple manner. Rather than discussing one specific instruction for one specific machine, we abstract the main concepts behind these types of instructions.

The `TestAndSet()` instruction can be defined as shown in Figure 6.6. The important characteristic is that this instruction is executed atomically. Thus, if two `TestAndSet()` instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order. If the machine supports the `TestAndSet()` instruction, then we can implement mutual exclusion by declaring a Boolean variable `lock`, initialized to `false`. The structure of process P_i is shown in Figure 6.7.

The `Swap()` instruction, in contrast to the `TestAndSet()` instruction, operates on the contents of two words; it is defined as shown in Figure 6.8. Like the `TestAndSet()` instruction, it is executed atomically. If the machine supports the `Swap()` instruction, then mutual exclusion can be provided as follows. A global Boolean variable `lock` is declared and is initialized to `false`. In addition, each process has a local Boolean variable `key`. The structure of process P_i is shown in Figure 6.9.

Although these algorithms satisfy the mutual-exclusion requirement, they do not satisfy the bounded-waiting requirement. In Figure 6.10, we present

```
void Swap(boolean *a, boolean *b) {
   boolean temp = *a;
   *a = *b;
   *b = temp;
}
```

Figure 6.8 The definition of the `Swap()` instruction.

```
do {
   key = TRUE;
   while (key == TRUE)
      Swap(&lock, &key);

      // critical section

   lock = FALSE;

      // remainder section
}while (TRUE);
```

Figure 6.9 Mutual-exclusion implementation with the `Swap()` instruction.

another algorithm using the `TestAndSet()` instruction that satisfies all the critical-section requirements. The common data structures are

```
boolean waiting[n];
boolean lock;
```

These data structures are initialized to `false`. To prove that the mutual-exclusion requirement is met, we note that process P_i can enter its critical section only if either `waiting[i] == false` or `key == false`. The value of `key` can become `false` only if the `TestAndSet()` is executed. The first process to execute the `TestAndSet()` will find `key == false`; all others must

```
do {
   waiting[i] = TRUE;
   key = TRUE;
   while (waiting[i] && key)
      key = TestAndSet(&lock);
   waiting[i] = FALSE;

      // critical section

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = FALSE;
   else
      waiting[j] = FALSE;

      // remainder section
}while (TRUE);
```

Figure 6.10 Bounded-waiting mutual exclusion with `TestAndSet()`.

wait. The variable `waiting[i]` can become `false` only if another process leaves its critical section; only one `waiting[i]` is set to `false`, maintaining the mutual-exclusion requirement.

To prove that the progress requirement is met, we note that the arguments presented for mutual exclusion also apply here, since a process exiting the critical section either sets `lock` to `false` or sets `waiting[j]` to `false`. Both allow a process that is waiting to enter its critical section to proceed.

To prove that the bounded-waiting requirement is met, we note that, when a process leaves its critical section, it scans the array `waiting` in the cyclic ordering ($i + 1, i + 2, ..., n - 1, 0, ..., i - 1$). It designates the first process in this ordering that is in the entry section (`waiting[j] == true`) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within $n - 1$ turns.

Unfortunately for hardware designers, implementing atomic `TestAndSet()` instructions on multiprocessors is not a trivial task. Such implementations are discussed in books on computer architecture.

6.5 Semaphores

The various hardware-based solutions to the critical-section problem (using the `TestAndSet()` and `Swap()` instructions) presented in Section 6.4 are complicated for application programmers to use. To overcome this difficulty, we can use a synchronization tool called a **semaphore**.

A semaphore `S` is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: `wait()` and `signal()`. The `wait()` operation was originally termed `P` (from the Dutch *proberen*, "to test"); `signal()` was originally called `V` (from *verhogen*, "to increment"). The definition of `wait()` is as follows:

```
wait(S) {
    while S <= 0
       ; // no-op
    S--;
}
```

The definition of `signal()` is as follows:

```
signal(S) {
    S++;
}
```

All the modifications to the integer value of the semaphore in the `wait()` and `signal()` operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of `wait(S)`, the testing of the integer value of `S` ($S \leq 0$), and its possible modification (`S--`), must also be executed without interruption. We shall see how these operations can be implemented in Section 6.5.2; first, let us see how semaphores can be used.

```
do {
   waiting(mutex);

      // critical section

   signal(mutex);

      // remainder section
}while (TRUE);
```

Figure 6.11 Mutual-exclusion implementation with semaphores.

6.5.1 Usage

Operating systems often distinguish between counting and binary semaphores. The value of a **counting semaphore** can range over an unrestricted domain. The value of a **binary semaphore** can range only between 0 and 1. On some systems, binary semaphores are known as **mutex locks**, as they are locks that provide *mut*ual *ex*clusion.

We can use binary semaphores to deal with the critical-section problem for multiple processes. The n processes share a semaphore, `mutex`, initialized to 1. Each process P_i is organized as shown in Figure 6.11.

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a `wait()` operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a `signal()` operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

We can also use semaphores to solve various synchronization problems. For example, consider two concurrently running processes: P_1 with a statement S_1 and P_2 with a statement S_2. Suppose we require that S_2 be executed only after S_1 has completed. We can implement this scheme readily by letting P_1 and P_2 share a common semaphore `synch`, initialized to 0, and by inserting the statements

```
$S_1$;
signal(synch);
```

in process P_1, and the statements

```
wait(synch);
$S_2$;
```

in process P_2. Because `synch` is initialized to 0, P_2 will execute S_2 only after P_1 has invoked `signal(synch)`, which is after statement S_1 has been executed.

6.5.2 Implementation

The main disadvantage of the semaphore definition given here is that it requires **busy waiting**. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a **spinlock** because the process "spins" while waiting for the lock. (Spinlocks do have an advantage in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. Thus, when locks are expected to be held for short times, spinlocks are useful; they are often employed on multiprocessor systems where one thread can "spin" on one processor while another thread performs its critical section on another processor.)

To overcome the need for busy waiting, we can modify the definition of the `wait()` and `signal()` semaphore operations. When a process executes the `wait()` operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can *block* itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

A process that is blocked, waiting on a semaphore `S`, should be restarted when some other process executes a `signal()` operation. The process is restarted by a `wakeup()` operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.)

To implement semaphores under this definition, we define a semaphore as a "C" struct:

```
typedef struct {
     int value;
     struct process *list;
} semaphore;
```

Each semaphore has an integer `value` and a list of processes `list`. When a process must wait on a semaphore, it is added to the list of processes. A `signal()` operation removes one process from the list of waiting processes and awakens that process.

The `wait()` semaphore operation can now be defined as

```
wait(semaphore *S) {
          S->value--;
          if (S->value < 0) {
                  add this process to S->list;
                  block();
          }
}
```

The signal() semaphore operation can now be defined as

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
               remove a process P from S->list;
               wakeup(P);
        }
}
```

The block() operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls.

Note that, although under the classical definition of semaphores with busy waiting the semaphore value is never negative, this implementation may have negative semaphore values. If the semaphore value is negative, its magnitude is the number of processes waiting on that semaphore. This fact results from switching the order of the decrement and the test in the implementation of the wait() operation.

The list of waiting processes can be easily implemented by a link field in each process control block (PCB). Each semaphore contains an integer value and a pointer to a list of PCBs. One way to add and remove processes from the list in a way that ensures bounded waiting is to use a FIFO queue, where the semaphore contains both head and tail pointers to the queue. In general, however, the list can use *any* queueing strategy. Correct usage of semaphores does not depend on a particular queueing strategy for the semaphore lists.

The critical aspect of semaphores is that they be executed atomically. We must guarantee that no two processes can execute wait() and signal() operations on the same semaphore at the same time. This is a critical-section problem; and in a single-processor environment (that is, where only one CPU exists), we can solve it by simply inhibiting interrupts during the time the wait() and signal() operations are executing. This scheme works in a single-processor environment because, once interrupts are inhibited, instructions from different processes cannot be interleaved. Only the currently running process executes until interrupts are reenabled and the scheduler can regain control.

In a multiprocessor environment, interrupts must be disabled on every processor; otherwise, instructions from different processes (running on different processors) may be interleaved in some arbitrary way. Disabling interrupts on every processor can be a difficult task and furthermore can seriously diminish performance. Therefore, SMP systems must provide alternative locking techniques—such as spinlocks—to ensure that wait() and signal() are performed atomically.

It is important to admit that we have not completely eliminated busy waiting with this definition of the wait() and signal() operations. Rather, we have removed busy waiting from the entry section to the critical sections of application programs. Furthermore, we have limited busy waiting to the critical sections of the wait() and signal() operations, and these sections are short (if properly coded, they should be no more than about ten instructions).

Thus, the critical section is almost never occupied, and busy waiting occurs rarely, and then for only a short time. An entirely different situation exists with application programs whose critical sections may be long (minutes or even hours) or may almost always be occupied. In such cases, busy waiting is extremely inefficient.

6.5.3 Deadlocks and Starvation

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a `signal()` operation. When such a state is reached, these processes are said to be **deadlocked**.

To illustrate this, we consider a system consisting of two processes, P_0 and P_1, each accessing two semaphores, S and Q, set to the value 1:

```
      P0               P1

  wait(S);         wait(Q);
  wait(Q);         wait(S);
     .                .
     .                .
     .                .
  signal(S);       signal(Q);
  signal(Q);       signal(S);
```

Suppose that P_0 executes `wait(S)` and then P_1 executes `wait(Q)`. When P_0 executes `wait(Q)`, it must wait until P_1 executes `signal(Q)`. Similarly, when P_1 executes `wait(S)`, it must wait until P_0 executes `signal(S)`. Since these `signal()` operations cannot be executed, P_0 and P_1 are deadlocked.

We say that a set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are *resource acquisition and release*. However, other types of events may result in deadlocks, as we shall show in Chapter 7. In that chapter, we shall describe various mechanisms for dealing with the deadlock problem.

Another problem related to deadlocks is **indefinite blocking**, or **starvation**, a situation in which processes wait indefinitely within the semaphore. Indefinite blocking may occur if we add and remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

6.6 Classic Problems of Synchronization

In this section, we present a number of synchronization problems as examples of a large class of concurrency-control problems. These problems are used for testing nearly every newly proposed synchronization scheme. In our solutions to the problems, we use semaphores for synchronization.

```
do {
    . . .
  // produce an item in nextp
    . . .
  wait(empty);
  wait(mutex);
    . . .
  // add nextp to buffer
    . . .
  signal(mutex);
  signal(full);
}while (TRUE);
```

Figure 6.12 The structure of the producer process.

6.6.1 The Bounded-Buffer Problem

The *bounded-buffer problem* was introduced in Section 6.1; it is commonly used to illustrate the power of synchronization primitives. We present here a general structure of this scheme without committing ourselves to any particular implementation; we provide a related programming project in the exercises at the end of the chapter.

We assume that the pool consists of *n* buffers, each capable of holding one item. The `mutex` semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The `empty` and `full` semaphores count the number of empty and full buffers. The semaphore `empty` is initialized to the value `n`; the semaphore `full` is initialized to the value 0.

The code for the producer process is shown in Figure 6.12; the code for the consumer process is shown in Figure 6.13. Note the symmetry between the producer and the consumer. We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

```
do {
  wait(full);
  wait(mutex);
    . . .
  // remove an item from buffer to nextc
    . . .
  signal(mutex);
  signal(empty);
    . . .
  // consume the item in nextc
    . . .
}while (TRUE);
```

Figure 6.13 The structure of the consumer process.

6.6.2 The Readers–Writers Problem

A database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. We distinguish between these two types of processes by referring to the former as **readers** and to the latter as **writers**. Obviously, if two readers access the shared data simultaneously, no adverse affects will result. However, if a writer and some other thread (either a reader or a writer) access the database simultaneously, chaos may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database. This synchronization problem is referred to as the *readers–writers problem*. Since it was originally stated, it has been used to test nearly every new synchronization primitive. The readers–writers problem has several variations, all involving priorities. The simplest one, referred to as the *first* readers–writers problem, requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting. The *second* readers–writers problem requires that, once a writer is ready, that writer performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve. For this reason, other variants of the problem have been proposed. In this section, we present a solution to the first readers–writers problem. Refer to the bibliographical notes at the end of the chapter for references describing starvation-free solutions to the second readers–writers problem.

In the solution to the first readers–writers problem, the reader processes share the following data structures:

```
semaphore mutex, wrt;
int readcount;
```

The semaphores `mutex` and `wrt` are initialized to 1; `readcount` is initialized to 0. The semaphore `wrt` is common to both reader and writer processes. The `mutex` semaphore is used to ensure mutual exclusion when the variable `readcount` is updated. The `readcount` variable keeps track of how many processes are currently reading the object. The semaphore `wrt` functions as a mutual-exclusion semaphore for the writers. It is also used by the first or last

```
do {
   wait(wrt);
      . . .
   // writing is performed
      . . .
   signal(wrt);
}while (TRUE);
```

Figure 6.14 The structure of a writer process.

```
do {
   wait(mutex);
   readcount++;
   if (readcount == 1)
      wait(wrt);
   signal(mutex);
      . . .
   // reading is performed
      . . .
   wait(mutex);
   readcount--;
   if (readcount == 0)
      signal(wrt);
   signal(mutex);
}while (TRUE);
```

Figure 6.15 The structure of a reader process.

reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical sections.

The code for a writer process is shown in Figure 6.14; the code for a reader process is shown in Figure 6.15. Note that, if a writer is in the critical section and n readers are waiting, then one reader is queued on `wrt`, and n − 1 readers are queued on `mutex`. Also observe that, when a writer executes `signal(wrt)`, we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

The readers–writers problem and its solutions has been generalized to provide **reader–writer** locks on some systems. Acquiring a reader–writer lock requires specifying the mode of the lock: either *read* or *write* access. When a process only wishes to read shared data, it requests the reader–writer lock in read mode; a process wishing to modify the shared data must request the lock in write mode. Multiple processes are permitted to concurrently acquire a reader–writer lock in read mode; only one process may acquire the lock for writing as exclusive access is required for writers.

Reader–writer locks are most useful in the following situations:

- In applications where it is easy to identify which processes only read shared data and which threads only write shared data.
- In applications that have more readers than writers. This is because reader–writer locks generally require more overhead to establish than semaphores or mutual exclusion locks, and the overhead for setting up a reader–writer lock is compensated by the increased concurrency of allowing multiple readers.

6.6.3 The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid

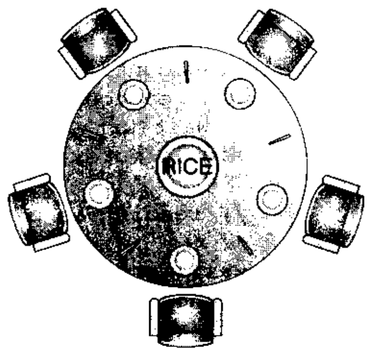


Figure 6.16 The situation of the dining philosophers.

with five single chopsticks (Figure 6.16). When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.

The *dining-philosophers problem* is considered a classic synchronization problem neither because of its practical importance nor because computer scientists dislike philosophers but because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a `wait()` operation on that semaphore; she releases her chopsticks by executing the `signal()` operation on the appropriate semaphores. Thus, the shared data are

```
semaphore chopstick[5];
```

where all the elements of `chopstick` are initialized to 1. The structure of philosopher *i* is shown in Figure 6.17.

Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock. Suppose that all five philosophers become hungry simultaneously and each grabs her left chopstick. All the elements of `chopstick` will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Several possible remedies to the deadlock problem are listed next. In Section 6.7, we present a solution to the dining-philosophers problem that ensures freedom from deadlocks.

- Allow at most four philosophers to be sitting simultaneously at the table.

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   // eat
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   // think
      . . .
}while (TRUE);
```

Figure 6.17 The structure of philosopher *i*.

Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this she must pick them up in a critical section).

Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

Finally, any satisfactory solution to the dining-philosophers problem must guard against the possibility that one of the philosophers will starve to death. A deadlock-free solution does not necessarily eliminate the possibility of starvation.

6.7 [illegible]

Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect, since these errors happen only if some particular execution sequences take place and these sequences do not always occur.

We have seen an example of such errors in the use of counters in our solution to the producer–consumer problem (Section 6.1). In that example, the timing problem happened only rarely, and even then the counter value appeared to be reasonable—off by only 1. Nevertheless, the solution is obviously not an acceptable one. It is for this reason that semaphores were introduced in the first place.

Unfortunately, such timing errors can still occur when semaphores are used. To illustrate how, we review the semaphore solution to the critical-section problem. All processes share a semaphore variable `mutex`, which is initialized to 1. Each process must execute `wait(mutex)` before entering the critical section and `signal(mutex)` afterward. If this sequence is not observed, two processes may be in their critical sections simultaneously. Let us examine the various difficulties that may result. Note that these difficulties will arise even if a *single* process is not well behaved. This situation may be caused by an honest programming error or an uncooperative programmer.

- Suppose that a process interchanges the order in which the `wait()` and `signal()` operations on the semaphore `mutex` are executed, resulting in the following execution:

```
signal(mutex);
   ...
   critical section
   ...
wait(mutex);
```

  In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement. This error may be discovered only if several processes are simultaneously active in their critical sections. Note that this situation may not always be reproducible.

- Suppose that a process replaces `signal(mutex)` with `wait(mutex)`. That is, it executes

```
wait(mutex);
   ...
   critical section
   ...
wait(mutex);
```

  In this case, a deadlock will occur.

- Suppose that a process omits the `wait(mutex)`, or the `signal(mutex)`, or both. In this case, either mutual exclusion is violated or a deadlock will occur.

These examples illustrate that various types of errors can be generated easily when programmers use semaphores incorrectly to solve the critical-section problem. Similar problems may arise in the other synchronization models that we discussed in Section 6.6.

To deal with such errors, researchers have developed high-level language constructs. In this section, we describe one fundamental high-level synchronization construct—the **monitor** type.

6.7.1 Usage

A type, or abstract data type, encapsulates private data with public methods to operate on that data. A monitor type presents a set of programmer-defined operations that are provided mutual exclusion within the monitor. The monitor type also contains the declaration of variables whose values define the state of an instance of that type, along with the bodies of procedures or functions that operate on those variables. The syntax of a monitor is shown in Figure 6.18. The representation of a monitor type cannot be used directly by the various processes. Thus, a procedure defined within a monitor can access only those variables declared locally within the monitor and its formal parameters. Similarly, the local variables of a monitor can be accessed by only the local procedures.

```
monitor monitor name
{
   // shared variable declarations

   procedure P1 ( . . . ) {
      . . .
   }

   procedure P2 ( . . . ) {
      . . .
   }
            .
            .
            .
   procedure Pn ( . . . ) {
      . . .
   }

   initialization code ( . . . ) {
      . . .
   }
}
```

Figure 6.18 Syntax of a monitor.

The monitor construct ensures that only one process at a time can be active within the monitor. Consequently, the programmer does not need to code this synchronization constraint explicitly (Figure 6.19). However, the monitor construct, as defined so far, is not sufficiently powerful for modeling some synchronization schemes. For this purpose, we need to define additional synchronization mechanisms. These mechanisms are provided by the `condition` construct. A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type *condition*:

```
condition x, y;
```

The only operations that can be invoked on a condition variable are `wait()` and `signal()`. The operation

```
x.wait();
```

means that the process invoking this operation is suspended until another process invokes

```
x.signal();
```

The `x.signal()` operation resumes exactly one suspended process. If no process is suspended, then the `signal()` operation has no effect; that is, the state of x is the same as if the operation had never been executed (Figure 6.20). Contrast this operation with the `signal()` operation associated with semaphores, which always affects the state of the semaphore.

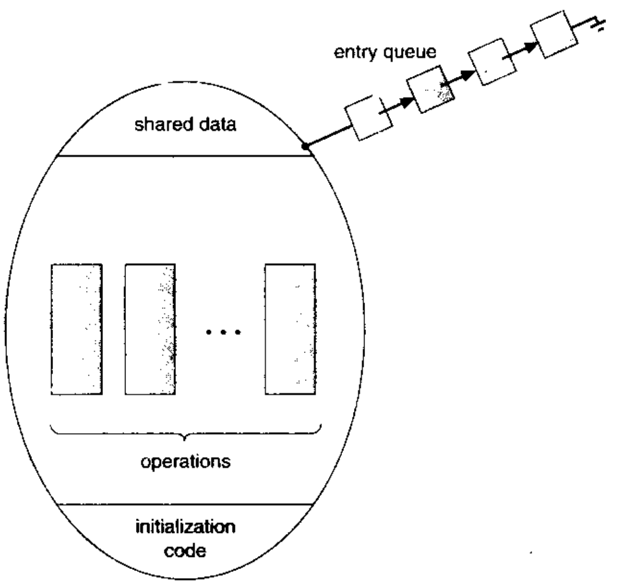


Figure 6.19 Schematic view of a monitor.

Now suppose that, when the `x.signal()` operation is invoked by a process P, there is a suspended process Q associated with condition x. Clearly, if the suspended process Q is allowed to resume its execution, the signaling process P must wait. Otherwise, both P and Q would be active simultaneously within the monitor. Note, however, that both processes can conceptually continue with their execution. Two possibilities exist:

Signal and wait. *P* either waits until *Q* leaves the monitor or waits for another condition.

Signal and continue. *Q* either waits until *P* leaves the monitor or waits for another condition.

There are reasonable arguments in favor of adopting either option. On the one hand, since *P* was already executing in the monitor, the *signal-and-continue* method seems more reasonable. On the other hand, if we allow thread *P* to continue, then by the time *Q* is resumed, the logical condition for which *Q* was waiting may no longer hold. A compromise between these two choices was adopted in the language Concurrent Pascal. When thread *P* executes the signal operation, it immediately leaves the monitor. Hence, *Q* is immediately resumed.

6.7.2 Dining-Philosophers Solution Using Monitors

We now illustrate monitor concepts by presenting a deadlock-free solution to the dining-philosophers problem. This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available. To

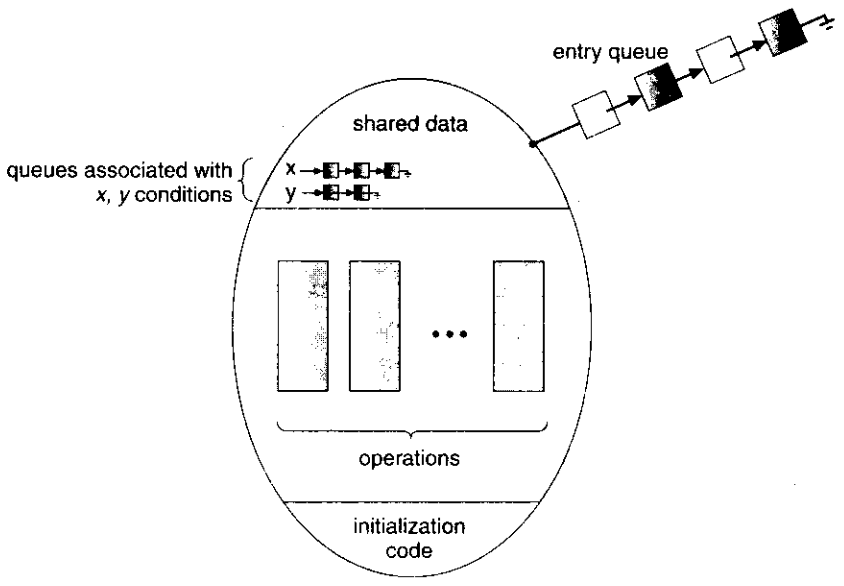


Figure 6.20 Monitor with condition variables.

code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:

```
enum {thinking, hungry, eating} state[5];
```

Philosopher *i* can set the variable `state[i] = eating` only if her two neighbors are not eating: (`state[(i+4) % 5] != eating`) and (`state[(i+1) % 5] != eating`).

We also need to declare

```
condition self[5];
```

where philosopher *i* can delay herself when she is hungry but is unable to obtain the chopsticks she needs.

We are now in a position to describe our solution to the dining-philosophers problem. The distribution of the chopsticks is controlled by the monitor `dp`, whose definition is shown in Figure 6.21. Each philosopher, before starting to eat, must invoke the operation `pickup()`. This may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the `putdown()` operation. Thus, philosopher *i* must invoke the operations `pickup()` and `putdown()` in the following sequence:

```
dp.pickup(i);
   ...
   eat
   ...
dp.putdown(i);
```

```
monitor dp
{
   enum {THINKING, HUNGRY, EATING}state[5];
   condition self[5];

   void pickup(int i) {
      state[i] = HUNGRY;
      test(i);
      if (state[i] != EATING)
         self[i].wait();
   }

   void putdown(int i) {
      state[i] = THINKING;
      test((i + 4) % 5);
      test((i + 1) % 5);
   }

   void test(int i) {
      if ((state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING)) {
           state[i] = EATING;
           self[i].signal();
      }
   }

   initialization_code() {
      for (int i = 0; i < 5; i++)
         state[i] = THINKING;
   }
}
```

Figure 6.21 A monitor solution to the dining-philosopher problem.

It is easy to show that this solution ensures that no two neighbors are eating simultaneously and that no deadlocks will occur. We note, however, that it is possible for a philosopher to starve to death. We do not present a solution to this problem but rather leave it as an exercise for you.

6.7.3 Implementing a Monitor Using Semaphores

We now consider a possible implementation of the monitor mechanism using semaphores. For each monitor, a semaphore `mutex` (initialized to 1) is provided. A process must execute `wait(mutex)` before entering the monitor and must execute `signal(mutex)` after leaving the monitor.

Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore, `next`, is introduced, initialized to 0, on which the signaling processes may suspend themselves. An integer variable next_count is also provided to count the number of processes suspended on next. Thus, each external procedure F is replaced by

```
wait(mutex);
   ...
   body of F
   ...
if (next_count > 0)
   signal(next);
else
   signal(mutex);
```

Mutual exclusion within a monitor is ensured.

We can now describe how condition variables are implemented. For each condition x, we introduce a semaphore x_sem and an integer variable x_count, both initialized to 0. The operation x.wait() can now be implemented as

```
x_count++;
if (next_count > 0)
   signal(next);
else
   signal(mutex);
wait(x_sem);
x_count--;
```

The operation x.signal() can be implemented as

```
if (x_count > 0) {
   next_count++;
   signal(x_sem);
   wait(next);
   next_count--;
}
```

This implementation is applicable to the definitions of monitors given by both Hoare and Brinch-Hansen. In some cases, however, the generality of the implementation is unnecessary, and a significant improvement in efficiency is possible. We leave this problem to you in Exercise 6.12.

6.7.4 Resuming Processes Within a Monitor

We turn now to the subject of process-resumption order within a monitor. If several processes are suspended on condition x, and an x.signal() operation is executed by some process, then how do we determine which of the suspended processes should be resumed next? One simple solution is to use an FCFS ordering, so that the process waiting the longest is resumed first. In many circumstances, however, such a simple scheduling scheme is not adequate. For this purpose, the **conditional-wait** construct can be used; it has the form

```
x.wait(c);
```

```
monitor ResourceAllocator
{
   boolean busy;
   condition x;

   void acquire(int time) {
      if (busy)
         x.wait(time);
      busy = TRUE;
   }

   void release() {
      busy = FALSE;
      x.signal();
   }

   initialization_code() {
      busy = FALSE;
   }
}
```

Figure 6.22 A monitor to allocate a single resource.

where c is an integer expression that is evaluated when the `wait()` operation is executed. The value of c, which is called a **priority number**, is then stored with the name of the process that is suspended. When `x.signal()` is executed, the process with the smallest associated priority number is resumed next.

To illustrate this new mechanism, we consider the `ResourceAllocator` monitor shown in Figure 6.22, which controls the allocation of a single resource among competing processes. Each process, when requesting an allocation of this resource, specifies the maximum time it plans to use the resource. The monitor allocates the resource to the process that has the shortest time-allocation request. A process that needs to access the resource in question must observe the following sequence:

```
R.acquire(t);
   ...
   access the resource;
   ...
R.release();
```

where R is an instance of type `ResourceAllocator`.

Unfortunately, the monitor concept cannot guarantee that the preceding access sequence will be observed. In particular, the following problems can occur:

- A process might access a resource without first gaining access permission to the resource.

- A process might never release a resource once it has been granted access [illegible]

- A process might attempt to release a resource that it never requested.
- A process might request the same resource twice (without first releasing the resource).

The same difficulties are encountered with the use of semaphores, and these difficulties are similar in nature to those that encouraged us to develop the monitor constructs in the first place. Previously, we had to worry about the correct use of semaphores. Now, we have to worry about the correct use of higher-level programmer-defined operations, with which the compiler can no longer assist us.

One possible solution to the current problem is to include the resource-access operations within the `ResourceAllocator` monitor. However, using this solution will mean that scheduling is done according to the built-in monitor-scheduling algorithm rather than the one we have coded.

To ensure that the processes observe the appropriate sequences, we must inspect all the programs that make use of the `ResourceAllocator` monitor and its managed resource. We must check two conditions to establish the correctness of this system. First, user processes must always make their calls on the monitor in a correct sequence. Second, we must be sure that an uncooperative process does not simply ignore the mutual-exclusion gateway provided by the monitor and try to access the shared resource directly, without using the access protocols. Only if these two conditions can be ensured can we guarantee that no time-dependent errors will occur and that the scheduling algorithm will not be defeated.

Although this inspection may be possible for a small, static system, it is not reasonable for a large system or a dynamic system. This access-control problem can be solved only by additional mechanisms that will be described in Chapter 17.

Many programming languages have incorporated the idea of the monitor as described in this section, including Concurrent Pascal, Mesa, C# (pronounced *C-sharp*), and Java. Other languages—such as Erlang—provide some type of concurrency support using a similar mechanism.

6.8 [illegible]

We next describe the synchronization mechanisms provided by the Solaris, Windows XP, and Linux operating systems, as well as the Pthreads API. We have chosen these three systems because they provide good examples of different approaches for synchronizing the kernel, and we have included the Pthreads API because it is widely used for thread creation and synchronization by developers on UNIX and Linux systems. As you will see in this section, the synchronization methods available in these differing systems vary in subtle and significant ways.

6.8.1 Synchronization in Solaris

To control access to critical sections, Solaris provides adaptive mutexes, condition variables, semaphores, reader–writer locks, and turnstiles. Solaris implements semaphores and condition variables essentially as they are presented in Sections 6.5 and 6.7. In this section, we describe adaptive mutexes, reader–writer locks, and turnstiles.

An **adaptive mutex** protects access to every critical data item. On a multiprocessor system, an adaptive mutex starts as a standard semaphore implemented as a spinlock. If the data are locked and therefore already in use, the adaptive mutex does one of two things. If the lock is held by a thread that is currently running on another CPU, the thread spins while waiting for the lock to become available, because the thread holding the lock is likely to finish soon. If the thread holding the lock is not currently in run state, the thread blocks, going to sleep until it is awakened by the release of the lock. It is put to sleep so that it will not spin while waiting, since the lock will not be freed very soon. A lock held by a sleeping thread is likely to be in this category. On a single-processor system, the thread holding the lock is never running if the lock is being tested by another thread, because only one thread can run at a time. Therefore, on this type of system, threads always sleep rather than spin if they encounter a lock.

Solaris uses the adaptive-mutex method to protect only data that are accessed by short code segments. That is, a mutex is used if a lock will be held for less than a few hundred instructions. If the code segment is longer than that, spin waiting will be exceedingly inefficient. For these longer code segments, condition variables and semaphores are used. If the desired lock is already held, the thread issues a wait and sleeps. When a thread frees the lock, it issues a signal to the next sleeping thread in the queue. The extra cost of putting a thread to sleep and waking it, and of the associated context switches, is less than the cost of wasting several hundred instructions waiting in a spinlock.

Reader–writer locks are used to protect data that are accessed frequently but are usually accessed in a read-only manner. In these circumstances, reader–writer locks are more efficient than semaphores, because multiple threads can read data concurrently, whereas semaphores always serialize access to the data. Reader–writer locks are relatively expensive to implement, so again they are used on only long sections of code.

Solaris uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or a reader–writer lock. A **turnstile** is a queue structure containing threads blocked on a lock. For example, if one thread currently owns the lock for a synchronized object, all other threads trying to acquire the lock will block and enter the turnstile for that lock. When the lock is released, the kernel selects a thread from the turnstile as the next owner of the lock. Each synchronized object with at least one thread blocked on the object's lock requires a separate turnstile. However, rather than associating a turnstile with each synchronized object, Solaris gives each kernel thread its own turnstile. Because a thread can be blocked only on one object at a time, this is more efficient than having a turnstile per object.

The turnstile for the first thread to block on a synchronized object becomes the turnstile for the object itself. Subsequent threads blocking on the lock will be added to this turnstile. When the initial thread ultimately releases the lock, it gains a new turnstile from a list of free turnstiles maintained by the kernel. To prevent a **priority inversion**, turnstiles are organized according to a **priority-inheritance protocol** (Section 19.5). This means that if a lower-priority thread currently holds a lock that a higher-priority thread is blocked on, the thread with the lower priority will temporarily inherit the priority of the higher-priority thread. Upon releasing the lock, the thread will revert to its original priority.

Note that the locking mechanisms used by the kernel are implemented for user-level threads as well, so the same types of locks are available inside and outside the kernel. A crucial implementation difference is the priority-inheritance protocol. Kernel-locking routines adhere to the kernel priority-inheritance methods used by the scheduler, as described in Section 19.5; user-level thread-locking mechanisms do not provide this functionality.

To optimize Solaris performance, developers have refined and fine-tuned the locking methods. Because locks are used frequently and typically are used for crucial kernel functions, tuning their implementation and use can produce great performance gains.

6.8.2 Synchronization in Windows XP

The Windows XP operating system is a multithreaded kernel that provides support for real-time applications and multiple processors. When the Windows XP kernel accesses a global resource on a uniprocessor system, it temporarily masks interrupts for all interrupt handlers that may also access the global resource. On a multiprocessor system, Windows XP protects access to global resources using spinlocks. Just as in Solaris, the kernel uses spinlocks only to protect short code segments. Furthermore, for reasons of efficiency, the kernel ensures that a thread will never be preempted while holding a spinlock.

For thread synchronization outside the kernel, Windows XP provides **dispatcher objects**. Using a dispatcher object, threads synchronize according to several different mechanisms, including mutexes, semaphores, events, and timers. The system protects shared data by requiring a thread to gain ownership of a mutex to access the data and to release ownership when it is finished. Semaphores behave as described in Section 6.5. **Events** are similar to condition variables; that is, they may notify a waiting thread when a desired condition occurs. Finally, timers are used to notify one (or more than one) thread that a specified amount of time has expired.

Dispatcher objects may be in either a signaled state or a nonsignaled state. A **signaled state** indicates that an object is available and a thread will not block when acquiring the object. A **nonsignaled state** indicates that an object is not available and a thread will block when attempting to acquire the object. We illustrate the state transitions of a mutex lock dispatcher object in Figure 6.23.

A relationship exists between the state of a dispatcher object and the state of a thread. When a thread blocks on a nonsignaled dispatcher object, its state changes from ready to waiting, and the thread is placed in a waiting queue

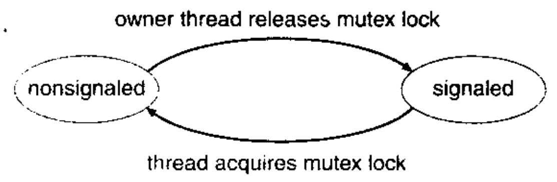


Figure 6.23 Mutex dispatcher object.

for that object. When the state for the dispatcher object moves to signaled, the kernel checks whether any threads are waiting on the object. If so, the kernel moves one thread—or possibly more threads—from the waiting state to the ready state, where they can resume executing. The number of threads the kernel selects from the waiting queue depends on the type of dispatcher object it is waiting on. The kernel will select only one thread from the waiting queue for a mutex, since a mutex object may be "owned" by only a single thread. For an event object, the kernel will select all threads that are waiting for the event.

We can use a mutex lock as an illustration of dispatcher objects and thread states. If a thread tries to acquire a mutex dispatcher object that is in a nonsignaled state, that thread will be suspended and placed in a waiting queue for the mutex object. When the mutex moves to the signaled state (because another thread has released the lock on the mutex), the thread waiting at the front of the queue will be moved from the waiting state to the ready state and will acquire the mutex lock.

We provide a programming project at the end of this chapter that uses mutex locks and semaphores in the Win32 API.

6.8.3 Synchronization in Linux

Prior to version 2.6, Linux was a nonpreemptive kernel, meaning that a process running in kernel mode could not be preempted—even if a higher-priority process became available to run. Now, however, the Linux kernel is fully preemptive, so a task can be preempted when it is running in the kernel.

The Linux kernel provides spinlocks and semaphores (as well as reader-writer versions of these two locks) for locking in the kernel. On SMP machines, the fundamental locking mechanism is a spinlock, and the kernel is designed so that the spinlock is held only for short durations. On single-processor machines, spinlocks are inappropriate for use and are replaced by enabling and disabling kernel preemption. That is, on single-processor machines, rather than holding a spinlock, the kernel disables kernel preemption; and rather than releasing the spinlock, it enables kernel preemption. This is summarized below:

single processor	multiple processors
Disable kernel preemption.	Acquire spin lock.
Enable kernel preemption.	Release spin lock.

Linux uses an interesting approach to disable and enable kernel preemption. It provides two simple system calls—`preempt disable()` and `preempt_enable()`—for disabling and enabling kernel preemption. In addition, however, the kernel is not preemptible if a kernel-mode task is holding a lock. To enforce this, each task in the system has a `thread-info` structure containing a counter, `preempt_count`, to indicate the number of locks being held by the task. When a lock is acquired, `preempt_count` is incremented. It is decremented when a lock is released. If the value of `preempt_count` for the task currently running is greater than zero, it is not safe to preempt the kernel, as this task currently holds a lock. If the count is zero, the kernel can safely be interrupted (assuming there are no outstanding calls to `preempt_disable()`).

Spinlocks—along with enabling and disabling kernel preemption—are used in the kernel only when a lock (or disabling kernel preemption) is held for a short duration. When a lock must be held for a longer period, semaphores are appropriate for use.

6.8.4 Synchronization in Pthreads

The Pthreads API provides mutex locks, condition variables, and read–write locks for thread synchronization. This API is available for programmers and is not part of any particular kernel. Mutex locks represent the fundamental synchronization technique used with Pthreads. A mutex lock is used to protect critical sections of code—that is, a thread acquires the lock before entering a critical section and releases it upon exiting the critical section. Condition variables in Pthreads behave much as described in Section 6.7. Read–write locks behave similarly to the locking mechanism described in Section 6.6.2. Many systems that implement Pthreads also provide semaphores, although they are not part of the Pthreads standard and instead belong to the POSIX SEM extension. Other extensions to the Pthreads API include spinlocks, although not all extensions are considered portable from one implementation to another. We provide a programming project at the end of this chapter that uses Pthreads mutex locks and semaphores.

6.9 [illegible]

The mutual exclusion of critical sections ensures that the critical sections are executed atomically. That is, if two critical sections are executed concurrently, the result is equivalent to their sequential execution in some unknown order. Although this property is useful in many application domains, in many cases we would like to make sure that a critical section forms a single logical unit of work that either is performed in its entirety or is not performed at all. An example is funds transfer, in which one account is debited and another is credited. Clearly, it is essential for data consistency either that both the credit and debit occur or that neither occur.

Consistency of data, along with storage and retrieval of data, is a concern often associated with **database systems**. Recently, there has been an upsurge of interest in using database-systems techniques in operating systems. Operating systems can be viewed as manipulators of data; as such, they can benefit from the advanced techniques and models available from database research. For instance, many of the ad hoc techniques used in operating systems to manage files could be more flexible and powerful if more formal database methods were used in their place. In Sections 6.9.2 to 6.9.4, we describe some of these database techniques and explain how they can be used by operating systems. First, however, we deal with the general issue of transaction atomicity. It is this property that the database techniques are meant to address.

6.9.1 System Model

A collection of instructions (or operations) that performs a single logical function is called a **transaction**. A major issue in processing transactions is the preservation of atomicity despite the possibility of failures within the computer system.

We can think of a transaction as a program unit that accesses and perhaps updates various data items that reside on a disk within some files. From our point of view, such a transaction is simply a sequence of `read` and `write` operations terminated by either a `commit` operation or an `abort` operation. A `commit` operation signifies that the transaction has terminated its execution successfully, whereas an `abort` operation signifies that the transaction has ended its normal execution due to some logical error or a system failure. If a terminated transaction has completed its execution successfully, it is **committed**; otherwise, it is **aborted**.

Since an aborted transaction may already have modified the data that it has accessed, the state of these data may not be the same as it would have been if the transaction had executed atomically. So that atomicity is ensured, an aborted transaction must have no effect on the state of the data that it has already modified. Thus, the state of the data accessed by an aborted transaction must be restored to what it was just before the transaction started executing. We say that such a transaction has been **rolled back**. It is part of the responsibility of the system to ensure this property.

To determine how the system should ensure atomicity, we need first to identify the properties of devices used for storing the various data accessed by the transactions. Various types of storage media are distinguished by their relative speed, capacity, and resilience to failure.

Volatile storage. Information residing in volatile storage does not usually survive system crashes. Examples of such storage are main and cache memory. Access to volatile storage is extremely fast, both because of the speed of the memory access itself and because it is possible to access directly any data item in volatile storage.

Nonvolatile storage. Information residing in nonvolatile storage usually survives system crashes. Examples of media for such storage are disks and magnetic tapes. Disks are more reliable than main memory but less reliable than magnetic tapes. Both disks and tapes, however, are subject to failure, which may result in loss of information. Currently, nonvolatile storage is slower than volatile storage by several orders of magnitude, because disk and tape devices are electromechanical and require physical motion to access data.

Stable storage. Information residing in stable storage is *never* lost (*never* should be taken with a grain of salt, since theoretically such absolutes cannot be guaranteed). To implement an approximation of such storage, we need to replicate information in several nonvolatile storage caches (usually disk) with independent failure modes and to update the information in a controlled manner (Section 12.8).

Here, we are concerned only with ensuring transaction atomicity in an environment where failures result in the loss of information on volatile storage.

6.9.2 Log-Based Recovery

One way to ensure atomicity is to record, on stable storage, information describing all the modifications made by the transaction to the various data it accesses. The most widely used method for achieving this form of recording is **write-ahead logging**. Here, the system maintains, on stable storage, a data structure called the **log**. Each log record describes a single operation of a transaction write and has the following fields:

- **Transaction name**. The unique name of the transaction that performed the `write` operation
- **Data item name**. The unique name of the data item written
- **Old value**. The value of the data item prior to the `write` operation
- **New value**. The value that the data item will have after the write

Other special log records exist to record significant events during transaction processing, such as the start of a transaction and the commit or abort of a transaction.

Before a transaction T_i starts its execution, the record $<$ T_i `starts`$>$ is written to the log. During its execution, any `write` operation by T_i is *preceded* by the writing of the appropriate new record to the log. When T_i commits, the record $<$ T_i `commits`$>$ is written to the log.

Because the information in the log is used in reconstructing the state of the data items accessed by the various transactions, we cannot allow the actual update to a data item to take place before the corresponding log record is written out to stable storage. We therefore require that, prior to execution of a `write`(X) operation, the log records corresponding to X be written onto stable storage.

Note the performance penalty inherent in this system. Two physical writes are required for every logical write requested. Also, more storage is needed, both for the data themselves and for the log recording the changes. In cases where the data are extremely important and fast failure recovery is necessary, the price is worth the functionality.

Using the log, the system can handle any failure that does not result in the loss of information on nonvolatile storage. The recovery algorithm uses two procedures:

- `undo`(T_i), which restores the value of all data updated by transaction T_i to the old values
- `redo`(T_i), which sets the value of all data updated by transaction T_i to the new values

The set of data updated by T_i and their respective old and new values can be found in the log.

The `undo` and `redo` operations must be idempotent (that is, multiple executions must have the same result as does one execution) to guarantee correct behavior, even if a failure occurs during the recovery process.

If a transaction T_i aborts, then we can restore the state of the data that it has updated by simply executing `undo`(T_i). If a system failure occurs, we restore the state of all updated data by consulting the log to determine which transactions need to be redone and which need to be undone. This classification of transactions is accomplished as follows:

- Transaction T_i needs to be undone if the log contains the $<$ T_i `starts`$>$ record but does not contain the $<$ T_i `commits`$>$ record.

Transaction T_i needs to be redone if the log contains both the $< T_i$ `starts`$>$ and the $< T_i$ `commits`$>$ records.

6.9.3 Checkpoints

When a system failure occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone. In principle, we need to search the entire log to make these determinations. There are two major drawbacks to this approach:

- The searching process is time consuming.
- Most of the transactions that, according to our algorithm, need to be redone have already actually updated the data that the log says they need to modify. Although redoing the data modifications will cause no harm (due to idempotency), it will nevertheless cause recovery to take longer.

To reduce these types of overhead, we introduce the concept of **checkpoints**. During execution, the system maintains the write-ahead log. In addition, the system periodically performs checkpoints that require the following sequence of actions to take place:

- Output all log records currently residing in volatile storage (usually main memory) onto stable storage.
- Output all modified data residing in volatile storage to the stable storage.
- Output a log record <`checkpoint`> onto stable storage.

The presence of a <`checkpoint`> record in the log allows the system to streamline its recovery procedure. Consider a transaction T_i that committed prior to the checkpoint. The $< T_i$ `commits`$>$ record appears in the log before the <`checkpoint`> record. Any modifications made by T_i must have been written to stable storage either prior to the checkpoint or as part of the checkpoint itself. Thus, at recovery time, there is no need to perform a `redo` operation on T_i.

This observation allows us to refine our previous recovery algorithm. After a failure has occurred, the recovery routine examines the log to determine the most recent transaction T_i that started executing before the most recent checkpoint took place. It finds such a transaction by searching the log backward to find the first <`checkpoint`> record, and then finding the subsequent $< T_i$ `start`$>$ record.

Once transaction T_i has been identified, the `redo` and `undo` operations need be applied only to transaction T_i and all transactions T_j that started executing after transaction T_i. We'll call these transactions set T. The remainder of the log can thus be ignored. The recovery operations that are required are as follows:

- For all transactions T_k in T such that the record $< T_k$ `commits`$>$ appears in the log, execute `redo`(T_k).

For all transactions T_k in T that have no $< T_k$ `commits`$>$ record in the log, execute undo(T_k).

6.9.4 Concurrent Atomic Transactions

We have been considering an environment in which only one transaction can be executing at a time. We now turn to the case where multiple transactions are active simultaneously. Because each transaction is atomic, the concurrent execution of transactions must be equivalent to the case where these transactions are executed serially in some arbitrary order. This property, called **serializability**, can be maintained by simply executing each transaction within a critical section. That is, all transactions share a common semaphore *mutex*, which is initialized to 1. When a transaction starts executing, its first action is to execute `wait`(*mutex*). After the transaction either commits or aborts, it executes `signal`(*mutex*).

Although this scheme ensures the atomicity of all concurrently executing transactions, it is nevertheless too restrictive. As we shall see, in many cases we can allow transactions to overlap their execution while maintaining serializability. A number of different **concurrency-control algorithms** ensure serializability. These algorithms are described below.

6.9.4.1 Serializability

Consider a system with two data items, A and B, that are both read and written by two transactions, T_0 and T_1. Suppose that these transactions are executed atomically in the order T_0 followed by T_1. This execution sequence, which is called a **schedule**, is represented in Figure 6.24. In schedule 1 of Figure 6.24, the sequence of instruction steps is in chronological order from top to bottom, with instructions of T_0 appearing in the left column and instructions of T_1 appearing in the right column.

A schedule in which each transaction is executed atomically is called a **serial schedule**. A serial schedule consists of a sequence of instructions from various transactions wherein the instructions belonging to a particular transaction appear together. Thus, for a set of n transactions, there exist $n!$ different valid serial schedules. Each serial schedule is correct, because it is

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Figure 6.24 Schedule 1: A serial schedule in which T_0 is followed by T_1.

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Figure 6.25 Schedule 2: A concurrent serializable schedule.

equivalent to the atomic execution of the various participating transactions in some arbitrary order.

If we allow the two transactions to overlap their execution, then the resulting schedule is no longer serial. A **nonserial schedule** does not necessarily imply an incorrect execution (that is, an execution that is not equivalent to one represented by a serial schedule). To see that this is the case, we need to define the notion of **conflicting operations**.

Consider a schedule S in which there are two consecutive operations O_i and O_j of transactions T_i and T_j, respectively. We say that O_i and O_j *conflict* if they access the same data item and at least one of them is a `write` operation. To illustrate the concept of conflicting operations, we consider the nonserial schedule 2 of Figure 6.25. The `write`(A) operation of T_0 conflicts with the `read`(A) operation of T_1. However, the `write`(A) operation of T_1 does not conflict with the `read`(B) operation of T_0, because the two operations access different data items.

Let O_i and O_j be consecutive operations of a schedule S. If O_i and O_j are operations of different transactions and O_i and O_j do not conflict, then we can swap the order of O_i and O_j to produce a new schedule S'. We expect S to be equivalent to S', as all operations appear in the same order in both schedules, except for O_i and O_j, whose order does not matter.

We can illustrate the swapping idea by considering again schedule 2 of Figure 6.25. As the `write`(A) operation of T_1 does not conflict with the `read`(B) operation of T_0, we can swap these operations to generate an equivalent schedule. Regardless of the initial system state, both schedules produce the same final system state. Continuing with this procedure of swapping nonconflicting operations, we get:

Swap the `read`(B) operation of T_0 with the `read`(A) operation of T_1.

Swap the `write`(B) operation of T_0 with the `write`(A) operation of T_1.

Swap the `write`(B) operation of T_0 with the `read`(A) operation of T_1.

The final result of these swaps is schedule 1 in Figure 6.24, which is a serial schedule. Thus, we have shown that schedule 2 is equivalent to a serial schedule. This result implies that, regardless of the initial system state, schedule 2 will produce the same final state as will some serial schedule.

If a schedule S can be transformed into a serial schedule S' by a series of swaps of nonconflicting operations, we say that a schedule S is **conflict serializable**. Thus, schedule 2 is conflict serializable, because it can be transformed into the serial schedule 1.

6.9.4.2 Locking Protocol

One way to ensure serializability is to associate with each data item a lock and to require that each transaction follow a **locking protocol** that governs how locks are acquired and released. There are various modes in which a data item can be locked. In this section, we restrict our attention to two modes:

- **Shared.** If a transaction T_i has obtained a shared-mode lock (denoted by S) on data item Q, then T_i can read this item but cannot write Q.
- **Exclusive.** If a transaction T_i has obtained an exclusive-mode lock (denoted by X) on data item Q, then T_i can both read and write Q.

We require that every transaction request a lock in an appropriate mode on data item Q, depending on the type of operations it will perform on Q.

To access data item Q, transaction T_i must first lock Q in the appropriate mode. If Q is not currently locked, then the lock is granted, and T_i can now access it. However, if the data item Q is currently locked by some other transaction, then T_i may have to wait. More specifically, suppose that T_i requests an exclusive lock on Q. In this case, T_i must wait until the lock on Q is released. If T_i requests a shared lock on Q, then T_i must wait if Q is locked in exclusive mode. Otherwise, it can obtain the lock and access Q. Notice that this scheme is quite similar to the readers–writers algorithm discussed in Section 6.6.2.

A transaction may unlock a data item that it locked at an earlier point. It must, however, hold a lock on a data item as long as it accesses that item. Moreover, it is not always desirable for a transaction to unlock a data item immediately after its last access of that data item, because serializability may not be ensured.

One protocol that ensures serializability is the **two-phase locking protocol**. This protocol requires that each transaction issue lock and unlock requests in two phases:

- **Growing phase.** A transaction may obtain locks but may not release any lock.
- **Shrinking phase.** A transaction may release locks but may not obtain any new locks.

Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and no more lock requests can be issued.

The two-phase locking protocol ensures conflict serializability (Exercise 6.17). It does not, however, ensure freedom from deadlock. In addition, it is possible that, for a given set of transactions, there are conflict-serializable schedules that cannot be obtained by use of the two-phase locking protocol. However, to improve performance over two-phase locking, we need either to have additional information about the transactions or to impose some structure or ordering on the set of data.

6.9.4.3 Timestamp-Based Protocols

In the locking protocols described above, the order followed by pairs of conflicting transactions is determined at execution time by the first lock that both request and that involves incompatible modes. Another method for determining the serializability order is to select an order in advance. The most common method for doing so is to use a **timestamp** ordering scheme.

With each transaction T_i in the system, we associate a unique fixed timestamp, denoted by TS(T_i). This timestamp is assigned by the system before the transaction T_i starts execution. If a transaction T_i has been assigned timestamp TS(T_i), and later a new transaction T_j enters the system, then TS(T_i) $<$ TS(T_j). There are two simple methods for implementing this scheme:

- Use the value of the system clock as the timestamp; that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system. This method will not work for transactions that occur on separate systems or for processors that do not share a clock.
- Use a logical counter as the timestamp; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system. The counter is incremented after a new timestamp is assigned.

The timestamps of the transactions determine the serializability order. Thus, if TS(T_i) $<$ TS(T_j), then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction T_i appears before transaction T_j.

To implement this scheme, we associate with each data item Q two timestamp values:

- **W-timestamp**(Q) denotes the largest timestamp of any transaction that successfully executed `write`(Q).
- **R-timestamp**(Q) denotes the largest timestamp of any transaction that successfully executed `read`(Q).

These timestamps are updated whenever a new `read`(Q) or `write`(Q) instruction is executed.

The timestamp-ordering protocol ensures that any conflicting `read` and `write` operations are executed in timestamp order. This protocol operates as follows:

Suppose that transaction T_i issues `read`(Q):

- If TS(T_i) $<$ W-timestamp(), then T_i needs to read a value of Q that was already overwritten. Hence, the `read` operation is rejected, and T_i is rolled back.
- If TS(T_i) $\geq$ W-timestamp(Q), then the `read` operation is executed, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(T_i).

Suppose that transaction T_i issues `write`(Q):

T_2	T_3
read(B)	
	read(B)
	write(B)
read(A)	
	read(A)
	write(A)

Figure 6.26 Schedule 3: A schedule possible under the timestamp protocol.

- If TS(T_i) < R-timestamp(Q), then the value of Q that T_i is producing was needed previously and T_i assumed that this value would never be produced. Hence, the `write` operation is rejected, and T_i is rolled back.
- If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q. Hence, this `write` operation is rejected, and T_i is rolled back.
- Otherwise, the `write` operation is executed.

A transaction T_i that is rolled back as a result of the issuing of either a `read` or `write` operation is assigned a new timestamp and is restarted.

To illustrate this protocol, consider schedule 3 of Figure 6.26, which includes transactions T_2 and T_3. We assume that a transaction is assigned a timestamp immediately before its first instruction. Thus, in schedule 3, TS(T_2) < TS(T_3), and the schedule is possible under the timestamp protocol.

This execution can also be produced by the two-phase locking protocol. However, some schedules are possible under the two-phase locking protocol but not under the timestamp protocol, and vice versa.

The timestamp protocol ensures conflict serializability. This capability follows from the fact that conflicting operations are processed in timestamp order. The protocol also ensures freedom from deadlock, because no transaction ever waits.

6.10

Given a collection of cooperating sequential processes that share data, mutual exclusion must be provided. One solution is to ensure that a critical section of code is in use by only one process or thread at a time. Different algorithms exist for solving the critical-section problem, with the assumption that only storage interlock is available.

The main disadvantage of these user-coded solutions is that they all require busy waiting. Semaphores overcome this difficulty. Semaphores can be used to solve various synchronization problems and can be implemented efficiently, especially if hardware support for atomic operations is available.

Various synchronization problems (such as the bounded-buffer problem, the readers–writers problem, and the dining-philosophers problem) are important mainly because they are examples of a large class of concurrency-control problems. These problems are used to test nearly every newly proposed synchronization scheme.

The operating system must provide the means to guard against timing errors. Several language constructs have been proposed to deal with these problems. Monitors provide the synchronization mechanism for sharing abstract data types. A condition variable provides a method by which a monitor procedure can block its execution until it is signaled to continue.

Operating systems also provide support for synchronization. For example, Solaris, Windows XP, and Linux provide mechanisms such as semaphores, mutexes, spinlocks, and condition variables to control access to shared data. The Pthreads API provides support for mutexes and condition variables.

A transaction is a program unit that must be executed atomically; that is, either all the operations associated with it are executed to completion, or none are performed. To ensure atomicity despite system failure, we can use a write-ahead log. All updates are recorded on the log, which is kept in stable storage. If a system crash occurs, the information in the log is used in restoring the state of the updated data items, which is accomplished by use of the undo and redo operations. To reduce the overhead in searching the log after a system failure has occurred, we can use a checkpoint scheme.

To ensure serializability when the execution of several transactions overlaps, we must use a concurrency-control scheme. Various concurrency-control schemes ensure serializability by delaying an operation or aborting the transaction that issued the operation. The most common ones are locking protocols and timestamp ordering schemes.

6.1 The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P_0 and P_1, share the following variables:

```
boolean flag[2]; /* initially false */
int turn;
```

The structure of process P_i (i == 0 or 1) is shown in Figure 6.27; the other process is P_j (j == 1 or 0). Prove that the algorithm satisfies all three requirements for the critical-section problem.

6.2 Explain why spinlocks are not appropriate for single-processor systems yet are often used in multiprocessor systems.

6.3 Explain why implementing synchronization primitives by disabling interrupts is not appropriate in a single-processor system if the synchronization primitives are to be used in user-level programs.

6.4 Describe how the `Swap()` instruction can be used to provide mutual exclusion that satisfies the bounded-waiting requirement.

6.5 Servers can be designed to limit the number of open connections. For example, a server may wish to have only N socket connections at any point in time. As soon as N connections are made, the server will not accept another incoming connection until an existing connection is released. Explain how semaphores can be used by a server to limit the number of concurrent connections.

```
do {
   flag[i] = TRUE;

   while (flag[j]) {
      if (turn == j) {
         flag[i] = false;
         while (turn == j)
            ; // do nothing
         flag[i] = TRUE;
      }
   }

      // critical section

   turn = j;
   flag[i] = FALSE;

      // remainder section
}while (TRUE);
```

Figure 6.27 The structure of process P_i in Dekker's algorithm.

6.6 The first known correct software solution to the critical-section problem for n processes with a lower bound on waiting of $n - 1$ turns was presented by Eisenberg and McGuire. The processes share the following variables:

```
enum pstate {idle, want_in, in_cs};
pstate flag[n];
int turn;
```

All the elements of `flag` are initially `idle`; the initial value of `turn` is immaterial (between 0 and `n-1`). The structure of process P_i is shown in Figure 6.28. Prove that the algorithm satisfies all three requirements for the critical-section problem.

6.7 Show how to implement the `wait()` and `signal()` semaphore operations in multiprocessor environments using the `TestAndSet()` instruction. The solution should exhibit minimal busy waiting.

6.8 **The Sleeping-Barber Problem**. A barbershop consists of a waiting room with n chairs and a barber room with one barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber. Write a program to coordinate the barber and the customers.

6.9 Write a bounded-buffer monitor in which the buffers (portions) are embedded within the monitor itself.

```
do {
   while (TRUE) {
      flag[i] = want_in;
      j = turn;

      while (j != i) {
         if (flag[j] != idle) {
            j = turn;
         else
            j = (j + 1) % n;
      }

      flag[i] = in_cs;
      j = 0;

      while ( (j < n) && (j == i || flag[j] != in_cs) )
         j++;

      if ( (j >= n) && (turn == i || flag[turn] == idle) )
         break;
   }

      // critical section

   j = (turn + 1) % n;

   while (flag[j] == idle)
      j = (j + 1) % n;

   turn = j;
   flag[i] = idle;

      // remainder section
}while (TRUE);
```

Figure 6.28 The structure of process P_i in Eisenberg and McGuire's algorithm.

6.10 The strict mutual exclusion within a monitor makes the bounded-buffer monitor of Exercise 6.9 mainly suitable for small portions.

a. Explain why this is true.

b. Design a new scheme that is suitable for larger portions.

6.11 How does the `signal()` operation associated with monitors differ from the corresponding operation defined for semaphores?

6.12 Suppose the `signal()` statement can appear only as the last statement in a monitor procedure. Suggest how the implementation described in Section 6.7 can be simplified.

6.13 A file is to be shared among different processes, each of which has a unique number. The file can be accessed simultaneously by several processes, subject to the following constraint: The sum of all unique numbers associated with all the processes currently accessing the file must be less than *n*. Write a monitor to coordinate access to the file.

6.14 When a signal is performed on a condition inside a monitor, the signaling process can either continue its execution or transfer control to the process that is signaled. How would the solution to the preceding exercise differ with the two different ways in which signaling can be performed?

6.15 Write a monitor that implements an *alarm clock* that enables a calling program to delay itself for a specified number of time units (*ticks*). You may assume the existence of a real hardware clock that invokes a procedure *tick* in your monitor at regular intervals.

6.16 Why do Solaris, Linux, and Windows 2000 use spinlocks as a synchronization mechanism only on multiprocessor systems and not on single-processor systems?

6.17 Show that the two-phase locking protocol ensures conflict serializability.

6.18 What are the implications of assigning a new timestamp to a transaction that is rolled back? How does the system process transactions that were issued after the rolled-back transaction but that have timestamps smaller than the new timestamp of the rolled-back transaction?

6.19 The `decrease_count()` function in the previous exercise currently returns 0 if sufficient resources are available and -1 otherwise. This leads to awkward programming for a process that wishes obtain a number of resources:

```
while (decrease_count(count) == -1)
   ;
```

Rewrite the resource-manager code segment using a monitor and condition variables so that the `decrease_count()` function suspends the process until sufficient resources are available. This will allow a process to invoke `decrease_count()` by simply calling

```
decrease_count(count);
```

The process will only return from this function call when sufficient resources are available.

In Section 6.6.1, we present a semaphore-based solution to the producer–consumer problem using a bounded buffer. In this project, we will design a programming solution to the bounded-buffer problem using the producer and consumer processes shown in Figures 6.12 and 6.13. The solution presented in Section 6.6.1 uses three semaphores: `empty` and `full`, which count the number of empty and full slots in the buffer, and mutex, which is a binary (or mutual exclusion) semaphore that protects the actual insertion or removal of items in the buffer. For this project, standard counting semaphores will be used for empty and full, and, rather than a binary semaphore, a mutex lock will be used to represent mutex. The producer and consumer—running as separate threads—will move items to and from a buffer that is synchronized with these empty, full, and mutex structures. You can solve this problem using either Pthreads or the Win32 API.

The Buffer

Internally, the buffer will consist of a fixed-size array of type buffer_item (which will be defined using a typedef). The array of buffer_item objects will be manipulated as a circular queue. The definition of buffer_item, along with the size of the buffer, can be stored in a header file such as the following:

```
/* buffer.h */
typedef int buffer_item;
#define BUFFER_SIZE 5
```

The buffer will be manipulated with two functions, insert_item() and remove_item(), which are called by the producer and consumer threads, respectively. A skeleton outlining these functions appears as:

```
#include <buffer.h>

/* the buffer */
buffer_item buffer[BUFFER_SIZE];

int insert_item(buffer_item item) {
   /* insert item into buffer
    return 0 if successful, otherwise
    return -1 indicating an error condition */
}

int remove_item(buffer_item *item) {
   /* remove an object from buffer
    placing it in item
    return 0 if successful, otherwise
    return -1 indicating an error condition */
}
```

The insert_item() and remove_item() functions will synchronize the producer and consumer using the algorithms outlined in Figures 6.12 and 6.13. The buffer will also require an initialization function that initializes the mutual-exclusion object mutex along with the empty and full semaphores.

The main() function will initialize the buffer and create the separate producer and consumer threads. Once it has created the producer and consumer threads, the main() function will sleep for a period of time and,

```
#include <buffer.h>

int main(int argc, char *argv[]) {
   /* 1. Get command line arguments argv[1], argv[2], argv[3] */
   /* 2. Initialize buffer */
   /* 3. Create producer thread(s) */
   /* 4. Create consumer thread(s) */
   /* 5. Sleep */
   /* 6. Exit */
}
```

Figure 6.29 A skeleton program

upon awakening, will terminate the application. The `main()` function will be passed three parameters on the command line:

How long to sleep before terminating

The number of producer threads

The number of consumer threads

A skeleton for this function appears as shown in Figure 6.29:

Producer and Consumer Threads

The producer thread will alternate between sleeping for a random period of time and inserting a random integer into the buffer. Random numbers will be produced using the `rand()` function, which produces random integers between 0 and RAND_MAX. The consumer will also sleep for a random period of time and, upon awakening, will attempt to remove an item from the buffer. An outline of the producer and consumer threads appears as shown in Figure 6.30. In the following sections, we first cover details specific to Pthreads and then describe details of the Win32 API.

Pthreads Thread Creation

Creating threads using the Pthreads API is discussed in Chapter 4. Please refer to that chapter for specific instructions regarding creation of the producer and consumer using Pthreads.

Pthreads Mutex Locks

In Figure 6.31 we show a code sample that illustrates how mutex locks available in the Pthread API can be used to protect a critical section.

Pthreads uses the `pthread_mutex_t` data type for mutex locks. A mutex is created with the `pthread_mutex_init(&mutex,NULL)` function, with the first parameter being a pointer to the mutex. By passing `NULL` as a second parameter, we initialize the mutex to its default attributes.

```
#include <stdlib.h> /* required for rand() */
#include <buffer.h>

void *producer(void *param) {
   buffer_item rand;

   while (TRUE) {
      /* sleep for a random period of time */
      sleep(...);
      /* generate a random number */
      rand = rand();
      printf("producer produced %f\n",rand);
      if (insert_item(rand))
         fprintf("report error condition");
}

void *consumer(void *param) {
   buffer_item rand;

   while (TRUE) {
      /* sleep for a random period of time */
      sleep(...);
      if (remove_item(&rand))
         fprintf("report error condition");
      else
         printf("consumer consumed %f\n",rand);
}
```

Figure 6.30 An outline of the producer and consumer threads.

The mutex is acquired and released with the pthread_mutex_lock() and pthread_mutex_unlock() functions. If the mutex lock is unavailable when pthread_mutex_lock() is invoked, the calling thread is blocked until the owner invokes pthread_mutex_unlock(). All mutex functions return a value of 0 with correct operation; if an error occurs, these functions return a nonzero error code.

Pthreads Semaphores

Pthreads provides two types of semaphores—named and unnamed. For this project, we use unnamed semaphores. The code below illustrates how a semaphore is created:

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 5 */
sem_init(&sem, 0, 5);
```

```
#include <pthread.h>
pthread_mutex_t mutex;

/* create the mutex lock */
pthread_mutex_init(&mutex,NULL);

/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/*** critical section ***/

/* release the mutex lock */
pthread_mutex unlock(&mutex);
```

Figure 6.31 fff

The `sem_init()` creates and initializes a semaphore. This function is passed three parameters:

- A pointer to the semaphore
- A flag indicating the level of sharing
- The semaphore's initial value

In this example, by passing the flag 0, we are indicating that this semaphore can only be shared by threads belonging to the same process that created the semaphore. A nonzero value would allow other processes to access the semaphore as well. In this example, we initialize the semaphore to the value 5.

In Section 6.5, we described the classical `wait()` and `signal()` semaphore operations. Pthreads names the `wait()` and `signal()` operations `sem_wait()` and `sem_post()`, respectively. The code example shown in Figure 6.32 creates a binary semaphore `mutex` with an initial value of 1 and illustrates its use in protecting a critical section.

Win32

Details concerning thread creation using the Win32 API are available in Chapter 4. Please refer to that chapter for specific instructions.

Win32 Mutex Locks

Mutex locks are a type of dispatcher object, as described in Section 6.8.2. The following illustrates how to create a mutex lock using the `CreateMutex()` function:

```
#include <windows.h>

HANDLE Mutex;
Mutex = CreateMutex(NULL, FALSE, NULL);
```

```
#include <semaphore.h>
sem_t sem mutex;

/* create the semaphore */
sem_init(&mutex, 0, 1);

/* acquire the semaphore */
sem_wait(&mutex);

/*** critical section ***/

/* release the semaphore */
sem_post(&mutex);
```

Figure 6.32 fff

The first parameter refers to a security attribute for the mutex lock. By setting this attribute to NULL, we are disallowing any children of the process creating this mutex lock to inherit the handle of the mutex. The second parameter indicates whether the creator of the mutex is the initial owner of the mutex lock. Passing a value of FALSE indicates that the thread creating the mutex is not the initial owner; we shall soon see how mutex locks are acquired. The third parameter allows naming of the mutex. However, because we provide a value of NULL, we do not name the mutex. If successful, `CreateMutex()` returns a HANDLE to the mutex lock; otherwise, it returns NULL.

In Section 6.8.2, we identified dispatcher objects as being either *signaled* or *nonsignaled*. A signaled object is available for ownership; once a dispatcher object (such as a mutex lock) is acquired, it moves to the nonsignaled state. When the object is released, it returns to signaled.

Mutex locks are acquired by invoking the `WaitForSingleObject()` function, passing the function the HANDLE to the lock and a flag indicating how long to wait. The following code demonstrates how the mutex lock created above can be acquired:

```
WaitForSingleObject(Mutex, INFINITE);
```

The parameter value INFINITE indicates that we will wait an infinite amount of time for the lock to become available. Other values could be used that would allow the calling thread to time out if the lock did not become available within a specified time. If the lock is in a signaled state, `WaitForSingleObject()` returns immediately, and the lock becomes nonsignaled. A lock is released (moves to the nonsignaled state) by invoking `ReleaseMutex()`, such as:

```
ReleaseMutex(Mutex);
```

Win32 Semaphores

Semaphores in the Win32 API are also dispatcher objects and thus use the same signaling mechanism as mutex locks. Semaphores are created as follows:

```
#include <windows.h>

HANDLE Sem;
Sem = CreateSemaphore(NULL, 1, 5, NULL);
```

The first and last parameters identify a security attribute and a name for the semaphore, similar to what was described for mutex locks. The second and third parameters indicate the initial value and maximum value of the semaphore. In this instance, the initial value of the semaphore is 1, and its maximum value is 5. If successful, `CreateSemaphore()` returns a `HANDLE` to the mutex lock; otherwise, it returns `NULL`.

Semaphores are acquired with the same `WaitForSingleObject()` function as mutex locks. We acquire the semaphore `Sem` created in this example by using the statement:

```
WaitForSingleObject(Semaphore, INFINITE);
```

If the value of the semaphore is > 0, the semaphore is in the signaled state and thus is acquired by the calling thread. Otherwise, the calling thread blocks indefinitely—as we are specifying `INFINITE`—until the semaphore becomes signaled.

The equivalent of the `signal()` operation on Win32 semaphores is the `ReleaseSemaphore()` function. This function is passed three parameters: (1) the HANDLE of the semaphore, (2) the amount by which to increase the value of the semaphore, and (3) a pointer to the previous value of the semaphore. We can increase `Sem` by 1 using the following statement:

```
ReleaseSemaphore(Sem, 1, NULL);
```

Both `ReleaseSemaphore()` and `ReleaseMutex()` return 0 if successful and nonzero otherwise.

Bibliographical Notes

The mutual-exclusion problem was first discussed in a classic paper by Dijkstra [1965a]. Dekker's algorithm (Exercise 6.1)—the first correct software solution to the two-process mutual-exclusion problem—was developed by the Dutch mathematician T. Dekker. This algorithm also was discussed by Dijkstra [1965a]. A simpler solution to the two-process mutual-exclusion problem has since been presented by Peterson [1981] (Figure 6.4).

Dijkstra [1965b] presented the first solution to the mutual-exclusion problem for n processes. This solution, however does not have an upper bound on the amount of time a process must wait before it is allowed to enter the critical section. Knuth [1966] presented the first algorithm with a bound; his bound was 2^n turns. A refinement of Knuth's algorithm by deBruijn [1967] reduced the waiting time to n^2 turns, after which Eisenberg and McGuire [1972] (Exercise 6.4) succeeded in reducing the time to the lower bound of $n-1$ turns. Another algorithm that also requires $n-1$ turns but is easier to program and to understand, is the bakery algorithm, which was developed by Lamport [1974]. Burns [1978] developed the hardware-solution algorithm that satisfies the bounded-waiting requirement.

General discussions concerning the mutual-exclusion problem were offered by Lamport [1986] and Lamport [1991]. A collection of algorithms for mutual exclusion was given by Raynal [1986].

The semaphore concept was suggested by Dijkstra [1965a]. Patil [1971] examined the question of whether semaphores can solve all possible synchronization problems. Parnas [1975] discussed some of the flaws in Patil's arguments. Kosaraju [1973] followed up on Patil's work to produce a problem that cannot be solved by `wait()` and `signal()` operations. Lipton [1974] discussed the limitations of various synchronization primitives.

The classic process-coordination problems that we have described are paradigms for a large class of concurrency-control problems. The bounded-buffer problem, the dining-philosophers problem, and the sleeping-barber problem (Exercise 6.8) were suggested by Dijkstra [1965a] and Dijkstra [1971]. The cigarette-smokers problem (Exercise 6.8) was developed by Patil [1971]. The readers–writers problem was suggested by Courtois et al. [1971]. The issue of concurrent reading and writing was discussed by Lamport [1977]. The problem of synchronization of independent processes was discussed by Lamport [1976].

The critical-region concept was suggested by Hoare [1972] and by Brinch-Hansen [1972]. The monitor concept was developed by Brinch-Hansen [1973]. A complete description of the monitor was given by Hoare [1974]. Kessels [1977] proposed an extension to the monitor to allow automatic signaling. Experience obtained from the use of monitors in concurrent programs was discussed in Lampson and Redell [1979]. General discussions concerning concurrent programming were offered by Ben-Ari [1990] and Birrell [1989].

Optimizing the performance of locking primitives has been discussed in many works, such as Lamport [1987], Mellor-Crummey and Scott [1991], and Anderson [1990]. The use of shared objects that do not require the use of critical sections was discussed in Herlihy [1993], Bershad [1993], and Kopetz and Reisinger [1993]. Novel hardware instructions and their utility in implementing synchronization primitives have been described in works such as Culler et al. [1998], Goodman et al. [1989], Barnes [1993], and Herlihy and Moss [1993].

Some details of the locking mechanisms used in Solaris were presented in Mauro and McDougall [2001]. Note that the locking mechanisms used by the kernel are implemented for user-level threads as well, so the same types of locks are available inside and outside the kernel. Details of Windows 2000 synchronization can be found in Solomon and Russinovich [2000].

The write-ahead log scheme was first introduced in System R by Gray et al. [1981]. The concept of serializability was formulated by Eswaran et al. [1976] in connection with their work on concurrency control for System R. The two-phase locking protocol was introduced by Eswaran et al. [1976]. The timestamp-based concurrency-control scheme was provided by Reed [1983]. An exposition of various timestamp-based concurrency-control algorithms was presented by Bernstein and Goodman [1980].