CHAPTER 5

Process and Thread Scheduling

5.1 ORGANIZATION OF SCHEDULERS
5.2 SCHEDULING METHODS
5.3 PRIORITY INVERSION
5.4 MULTIPROCESSOR AND DISTRIBUTED SCHEDULING

One of the major tasks performed by an operating system (OS) is to allocate ready processes or threads to the available processors. This task may usefully be divided into two parts. The first part, referred to as **process scheduling**, embodies decision-making policies to determine the order in which active processes should compete for the use of processors. The actual binding of a selected process to a processor, which involves removing the process from the ready queue, changing its status, and loading the processor state, is performed by a **process dispatcher**. Scheduling and dispatching are typically done by different modules of the OS. Frequently, these modules are part of the kernel, but they also could appear at higher OS levels and in applications code. In a more complex fashion, different elements of the system may even have their own schedulers so that scheduling is distributed throughout. Unless explicitly distinguished, we will refer to both components jointly as the **scheduler** and to both tasks jointly as **scheduling**.

The next section is concerned primarily with the general design of schedulers, with particular emphasis on priority-based systems. Following that is a detailed description and evaluation of common scheduling strategies used for batch, interactive, and real-time applications. The reminder of this chapter presents the priority inversion problem and discusses scheduling for multiprocessor and distributed systems.

5.1 ORGANIZATION OF SCHEDULERS

5.1.1 Embedded and Autonomous Schedulers

A scheduler may be shared by processes in the system in the same manner that the kernel operations developed in Chapter 4 are shared—the scheduler is invoked by a function or procedure call as an indirect result of a kernel operation. The kernel and the scheduler are then contained in the address space of all processes and execute within any process. They are *embedded* as functions or procedures of currently running processes.

A second method is to centralize the scheduler, possibly with a centralized kernel. Conceptually, this type of scheduler is considered a separate and *autonomous process*,...
FIGURE 5-1. Organization of schedulers. (a) shared scheduler (b) autonomous scheduler

running on its own processor. It can continuously poll the system for work, or it may be driven by wakeup signals. The autonomous and the shared alternatives are illustrated in Figure 5-1.

A shared approach offers a number of advantages. Because shared-kernel components are automatically included in every process, they are independent of the number of available processors and need not be permanently associated with any particular one. The view that the kernel really defines a set of machine operations necessary to transform a standard computer into an OS machine leads naturally to a shared organization. A final point is that different groups of processes may in principle have different OSs controlling them. Here, several different schedulers may coexist simultaneously, each attached to (shared by) a different set of processes.

The autonomous approach may be preferable in multiprocessor and distributed systems where one processor can be permanently dedicated to scheduling, kernel, and other supervisory activities such as I/O and program loading. In such a master/slave organization, the OS (i.e., the master) is, at least in principle, cleanly separated from user processes (i.e., the slaves). Even if a separate processor is not dedicated exclusively to executing parts of the OS, the autonomous scheduler approach is still possible in both uniprocessor and multiprocessor configurations. A logical difficulty, however, arises in these situations—namely, who dispatches the scheduler process? One solution is to automatically transfer control to the scheduler whenever a process blocks, is interrupted, or is awakened. In a time-sharing system, the scheduler is typically invoked at the end of a quantum interrupt, i.e., when a process has run for a predefined period time (i.e., the quantum) or has voluntarily given up the CPU. The scheduler dispatches the next process and resets the timer to count down the next quantum.
SECTION 5.1 Organization of Schedulers

1. **UNIX.** UNIX is an example of a system where the scheduler (dispatcher) is a separate process. It runs every other time, i.e., between any two other processes. It also serves as the “dummy” or idle process which runs when no other process is ready, as mentioned earlier in Section 3.5.1. A detailed example of a shared scheduler will be presented shortly.

2. **Windows 2000.** Windows 2000 uses an embedded scheduler (dispatcher), consisting of a set of routines invoked when any of the following events occur:
   - A new thread is created.
   - A previously blocked thread is awakened.
   - A thread uses up its time quantum.
   - A thread blocks itself.
   - A thread terminates.
   - A thread’s priority changes.
   - The preferred processor of a thread in a multiprocessor system (called the processor affinity) changes.

### 5.1.2 Priority Scheduling

Virtually all schedulers use the concept of priority when deciding which process should be running next. We will define the notion of priority formally in Section 5.2.1. For now, assume that priority is a numeric value, denoted as $P$ and derived for each process $p$ by a specific function $P = \text{Priority}(p)$. Depending on the function used, the value of $P$ may be static; i.e., it remains constant for the lifespan of the process. Or it can be dynamic, i.e., it is recomputed every time the scheduler is invoked. Regardless of how the priorities are derived for each process, they divide all processes into multiple priority levels. The priority levels are generally implemented as a data structure, referred to as the ready list ($RL$). Processes at a given level $RL[i]$ have precedence over all processes at lower levels $RL[j]$, $j < i$. Processes at the same priority level are ordered according to additional criteria, such as their order of arrival in the system. Section 4.2 presented several options for implementing priority queues.

The scheduler is invoked either periodically or as the results of events that can affect the priorities of the existing processes. The role of the scheduler is then to guarantee that, at the time it terminates, the $np$ highest-priority processes are running, where $np$ is the number of processors in the system. In other words, the priority of any running process must always be greater than or equal to that of any ready_a process.

We now present a concrete example of a priority scheduler/dispatcher that implements the above principles. The implementation follows the embedded design, where the Scheduler() function is invoked at the end of other kernel operations described in Chapter 4. These operations generally change the current state of the system. In particular, CPUs become inactive as a result of a Suspend, a Destroy, or a blocking operation, such as a $P$ operation on semaphores, requesting a currently unavailable resource; processes become ready_a, possibly resulting in preemption, due to a wakeup operation on a resource, such as a $V$ operation, or an Activate.
Chapter 5  Process and Thread Scheduling

The ready list \( RL \) is structured as a multilevel priority list, shown in Figure 4-3a; it contains all processes having \( \text{Status.Type} \in \{ \text{running, ready.a, ready.s} \} \). The basic algorithm of \( \text{Scheduler} \) is as follows:

\[
\text{Scheduler}() \{ \\
\text{do} \{ \\
\text{Find highest priority ready.a process } p; \\
\text{Find a free cpu;} \\
\text{if (cpu != NIL) } \text{Allocate.CPU}(p, \text{cpu}); \\
\} \text{ while (cpu != NIL);} \\
\text{do} \{ \\
\text{Find highest priority ready.a process } p; \\
\text{Find lowest priority running process } q; \\
\text{if (Priority}(p) > \text{Priority}(q)) \text{ Preempt}(p, q); \\
\} \text{ while (Priority}(p) > \text{Priority}(q)); \\
\text{if (self->Status.Type != 'running') } \text{Preempt}(p, \text{self}); \\
\}
\]

During the first do-loop, the scheduler allocates the highest-priority processes to free CPUs, if any are available. The function \( \text{Allocate.CPU} \) dispatches each process to the CPU and adjusts its descriptor to reflect its new \text{running} status. During the second do-loop, the scheduler tries to preempt running processes. This must happen whenever the priority of a currently running process is lower than the priority of another ready process; such a condition occurs when a new process is created or an existing one is unblocked.

Preemption is accomplished by the \( \text{Preempt} \) function which performs the same tasks as \( \text{Allocate.CPU} \) but does this after preempting the current (lower-priority) running process. This involves stopping the process and saving its current CPU state in its PCB (Fig. 4-5). The CPU state of the process to be restarted is loaded into the CPU instead.

Special care must be taken if the process to be preempted is the one currently running the scheduler. In this case, the actual preemption must be postponed to permit the scheduler to complete its task. Only when there are no other lower-priority running processes to be preempted does the running process give up the CPU by preempting itself.

There is one other condition under which preemption is necessary—namely when the state of the currently running process (self) is not “running.” This can happen, for example, when the scheduler is called from a request operation that could not be satisfied. As a result, the state of the invoking process was already changed to “blocked” by the time the scheduler function was invoked. When the scheduler finds this to be the case (last if-statement in the above code), the current process (the one running the scheduler) must give up the CPU.

The scheduler, while manageable, can nevertheless be simplified if some of the underlying assumptions are relaxed. The common situation in which only a single central processor is available \((np = 1)\) leads to a much less complex scheduling algorithm. In this case, neither \( \text{Suspend} \) nor \( \text{Destroy} \) would ever call \( \text{Scheduler} \); and, at most one process switch could occur. Another possibility is to limit resource allocators so that at most one process is ever taken off a blocked list (awakened) at any time, for example, by not supporting a \text{broadcast} version of a synchronization operation or by releasing resources one at a time.
5.2 SCHEDULING METHODS

In a typical state of execution, the number of active processes able to run will exceed the number of available processors. The task of a scheduler is to examine the current allocation of processes to processors and, when appropriate, perform a reallocation according to some scheduling policy. It is usually desirable to employ different strategies for different types of processes. For example, we might wish to discriminate between processes associated with batch jobs and those of interactive users, between system and user processes, or between processes dealing directly with I/O and those engaged in purely internal activities. With a general priority mechanism and the careful use of timers, a uniform organization can be designed to handle a broad spectrum of such strategies. We present a general framework for possible scheduling strategies, which formalizes the notion of priority scheduling introduced in the preceding section. Within this framework, we discuss a variety of common scheduling strategies. Such strategies can be incorporated into actual schedulers, such as the one presented in the last section.

5.2.1 A Framework for Scheduling

A scheduler can be specified within the following general framework (Ruschitzka and Fabry 1977), which asks two fundamental questions:

- **When** to schedule, i.e., when the scheduler should be invoked. This decision is governed by the **decision mode**.

- **Who** to schedule, i.e., how to choose a process (or thread) among the currently active processes. This is governed jointly by a **priority function** and an **arbitration rule**.

At certain points in time, specified by the decision mode, the scheduler evaluates the priority function for all active processes in the system. For each process, the priority function yields a value, referred to as the **current priority**. The processes with the highest priorities are assigned to the existing CPUs. The arbitration rule is applied in the case of ties when there is more than one process with the same priority. Depending on the particular decision mode, priority function, and arbitration rule, different scheduling disciplines may be defined.

The Decision Mode

The decision mode specifies the points in time at which process priorities are evaluated and compared, and at which processes are selected for execution. The allocation of processes to CPUs cannot change between two consecutive decision times.

There are two basic types of decision modes: **nonpreemptive** and **preemptive**. In the first case, a process, once started, is always allowed to run when logically possible; i.e., scheduling decisions are made only when a process terminates or blocks itself by requesting a service from the OS, or when a newly arriving process finds an idle CPU. Although they are more economical, nonpreemptive algorithms are usually not adequate in real-time or time-shared systems. Rather, a preemptive decision mode must be used. This permits a currently running process to be stopped and the CPU assigned to another process with higher priority.
The decision to attempt a preemption may be made when a new process arrives or when an existing process is awakened as a result of a message or interrupt. Preemption also may be performed periodically each time a process has executed for a period of $q$ milliseconds. In this case, the algorithm is called quantum-oriented, with $q$ as its quantum size. Preemption also may occur whenever the priority of a ready process rises above the priority of a currently running process. This could occur in systems that dynamically change the relative priorities of existing processes.

Preemptive scheduling policies are usually more costly than nonpreemptive ones because of process switching times, the additional logic in the scheduler itself, and the possible swapping of programs and data between primary and secondary memories. Many systems use a combination of both strategies. For example, a critical part of an OS process, such as a kernel routine, may be nonpreemptive, whereas most user processes are preemptive.

The Priority Function

The priority function when applied to a process $p$ generates a numeric value, $P$, that represents $p$’s current priority. The function may use different attributes of the process or the system as a whole to generate the value. The most common attributes applicable to a given process are the following:

- attained service time;
- real-time in system;
- total service time;
- deadline;
- periodicity;
- external priority;
- memory requirements.

The first three time-related parameters have two possible interpretations. In long-term scheduling, also referred to as job scheduling, where the scheduler’s task is to choose among possible batch jobs to be included in the ready list, the times relate to the entire run of the job (process). That means, its arrival in the system is defined as the point during which its status changed to ready for the first time, and its departure from the system is the time at which the process terminates or is destroyed.

In process scheduling, where the scheduler’s (dispatcher’s) task is to select among the currently ready processes to run next on the CPU, the times relate to the period between two consecutive blocking operations. That means, the arrival of a process is the time when its status changes from blocked to ready as the result of a wakeup operation. Its departure is the time when it status changes from running to blocked, as a result of a request operation. Between its arrival and departure time, the process’ status can switch many times between running and ready.

In this section, we are concerned primarily with process scheduling, but some of the algorithms presented are applicable also to long-term job scheduling.
Section 5.2  Scheduling Methods  153

- The **attained service time** is the amount of time during which the process was using the CPU since its arrival and is often the most important parameter for scheduling.

- The **real time in system** is the actual time the process has spent in the system since its arrival. It is composed of the **attained service time** and the **waiting time** during which the process was ready but not running. Note that the attained service time increases at the rate of the actual (real) time when the process is running and remains constant when the process is waiting for the CPU.

- The **total service time** is the CPU time the process will consume during its lifetime, i.e., before it departs (terminates or blocks on a request). Note that, at the point the process departs, its total service time is equal to the attained service time. Some systems assume that the total service time is known in advance. This is usually true of repetitive service tasks, such as the processing of a stream of signals or messages. In other cases, the total service time frequently can be predicted based on the most recent behavior of the process: A process that performs long computations without I/O or other systems services will block infrequently, resulting in long total service times. A process performing a lot of I/O operations will have short total service time requirements. Since processes tend to remain processor-bound or I/O-bound for significant periods of time, we can devise a function that predicts the next total service time period based on the most recent periods.

- A **deadline** is a point in real time by which a task must be completed. The need for such a requirement arises in situations where the results of a computation are time-sensitive in that their usefulness or value decreases with time. As an extreme example, the results of a weather forecast are useless after the date for which the prediction is made. Deadlines are specially important in real-time applications. For example, the transmission of a video or audio signal can be seriously distorted if deadlines for most data packets are not met. In safety- and mission-critical applications, computations that are not completed in time may lead to extremely costly or life-threatening situations. In these systems, priorities often correspond directly with deadlines; the shorter the deadline, the higher the priority.

- **Periodicity** refers to the repetition of a computation. In many real-time applications, the same, usually very short, computation is repeated at fixed intervals. Many other applications that are not normally considered as real-time also use periodic processes. One example is an email system that checks periodically for mail or message arrivals. Even event-driven processes that are not periodic are often handled as periodic processes by polling for events at a suitable frequency.

  A fixed period automatically imposes an implicit deadline: The current computation must be completed before the start of the next period. The converse, however, is not generally true. Deadlines may be imposed for reasons other than periodic computations.

- **External priorities** are values assigned explicitly to processes. They may be used as base priorities to differentiate between different classes of processes. One common policy, for example, is to assign the highest priorities to real-time processes,
the next lower priorities to interactive processes, and the lowest to batch processes. In a business organization, the priority of a user process may correspond to the user’s position in the company hierarchy. Historically, when computers were more expensive than people, a cost based on the type of resources employed was frequently associated with each priority level. This setup permits users to select the priority for their processes which provides adequate service (response) at a price they can afford. For example, a process with a given priority $P$ could be charged a dollar amount $f_1(P)$ for each second of processor time, $f_2(P)$ for each block of main memory per second, $f_3(P)$ for each second of terminal connect time, and $f_4(P)$ for each I/O operation.

- The memory requirements specify the amount of main memory the process needs for its execution. This serves as a major scheduling criterion in batch OSs. In interactive systems, it is also important since it is a good measure of swapping overhead. However, other criteria, typically those related to time, are usually more important.

In addition to the process-specific attributes discussed above, the priority function can consider various attributes of the system as a whole. Notably, the overall system load is an important parameter, since it has a significant effect on system response. Under heavy load, some schedulers would attempt to maintain good response to high-priority processes by discriminating more strongly among external priorities. Thus, the distinction between the “rich” and the “poor” becomes even more pronounced when resources become scarce. Others take a more egalitarian point of view by attempting to increase the overall system throughput, for example by reducing swapping overhead through the use of larger quantum sizes. This idea improves the general conditions for all participants uniformly.

**The Arbitration Rule**

This rule resolves conflicts among processes with equal priority. If the likelihood of two or more processes having the same priority is low because of the nature of the priority function, the choice could be performed at random. In quantum-oriented preemptive scheduling algorithms, processes with equal highest priority are usually assigned quanta in a cyclic (round-robin [RR]) manner. In most other schemes, the chronological ordering of process arrivals serves as the arbitration rule; i.e., processes with equal priority are dispatched on a FIFO basis.

### 5.2.2 Common Scheduling Algorithms

The scheduler framework presented in the previous section is very general: Many scheduling algorithms may be expressed by specifying a decision mode, a priority function of various parameters, and an arbitration rule. In the following two sections, we describe and compare the most widely used methods in terms of these three concepts that govern when and who to schedule. We will use a simple example to illustrate the main features of each algorithm. This assumes there are two processes, $p_1$ and $p_2$; $p_1$ arrives at time 0 and needs five units of CPU time to complete its task; $p_2$ arrives at time 2 and needs two units of CPU time. A scheduling decision is to be made at time 2. Figure 5-2 illustrates this scenario.
Section 5.2  Scheduling Methods  155

First-In/First-Out
The FIFO strategy dispatches processes according to their arrival time; the earlier the arrival, the higher the process priority. Thus, the current priority, \( P \), of a process depends only on the real time, \( r \), the process has spent in the system since its arrival: The larger the value of \( r \), the higher the process priority. The priority function then may be expressed as \( P = r \). The decision mode of FIFO is nonpreemptive, whereas its arbitration rule assumes a random choice among processes arriving at exactly the same time.

EXAMPLE: FIFO Algorithm
Assume that at time 2 (Fig. 5-2), neither process has been started yet. The priority of process \( p_1 \) at that time is \( r_{p_1} = 2 \); the priority of process \( p_2 \) is \( r_{p_2} = 0 \). Thus, \( p_1 \) starts, and it executes until it finishes its task. Then \( p_2 \) is allowed to run.

Shortest-Job-First
The shortest-job-first (SJF) policy is a nonpreemptive strategy that dispatches processes according to their total service time, \( t \). The priority varies inversely with the service time: The larger the value of \( t \), the lower the current priority. Thus, the priority function may be expressed as \( P = -t \). Either a chronological or a random ordering is normally employed as the arbitration rule for processes with the same time requirements.

EXAMPLE: SJF Algorithm
Assume again that at time 2, neither process has been started yet. The priority of process \( p_1 \) at that time is \( t_{p_1} = -5 \); the priority of process \( p_2 \) is \( t_{p_2} = -2 \). Thus, \( p_2 \) starts. However, if \( p_1 \) had already started at time 2, it would be allowed to finish its task before \( p_2 \) could run, since SJF is nonpreemptive.

Shortest-Remaining-Time
The shortest-remaining-time (SRT) policy may be viewed as the preemptive and dynamic version of SJF. It assigns the highest priority to the process that will need the least amount of time to complete, i.e., the remaining time \( t - a \). The priority is dynamic, increasing as

![Arrivals diagram](image-url)
a process runs and its time to completion becomes smaller. Thus, the priority function has the form \( P = -(t - a) \). The same arbitration rule as in SJF is usually employed.

**EXAMPLE: SRT Algorithm**

Assume that \( p_1 \) started executing immediately upon its arrival, i.e., at time 0. Then at time 2, the priorities of the two processes are as follows: The priority of \( p_1 \) is \(-2 \leq 3\), whereas the priority of \( p_2 \) is \( -2 \geq 0 \). Since \(-2 > 3\), \( p_2 \) preempts process \( p_1 \), which resumes its execution after \( p_2 \) terminates.

**Round-Robin**

The round-robin (RR) scheduling discipline imposes a fixed-time quantum \( q \), also referred to as a *time slice*, on the amount of *continuous* CPU time that may be used by any process. If a process has run for \( q \) continuous time units, it is preempted from its processor and placed at the end of the process list waiting for the processor. Note that all processes have the same priority, i.e., \( P = 0 \). The RR behavior is enforced by using a quantum-oriented decision mode that causes the assignment of processes to processors to be reevaluated every \( q \) time units. Since the priorities of all processes are the same, the cyclic arbitration rule is applied at that time.

The time quantum \( q \) may be constant for long periods of time, or it may vary with the process load. If the time \( Q \) for one complete RR of all active processes is kept constant, \( q \) can be computed dynamically every \( Q \) seconds as \( q = Q/n \), where \( n \) is the number of active processes competing for the CPU. Methods based on the RR scheme are used by most time-sharing systems for handling interactive processes.

**EXAMPLE: RR Algorithm**

Assume a time quantum of 0.1 time units. At time 2 in Figure 5-2, the CPU begins to switch between the two processes every 0.1 time units. Thus, both processes run concurrently but at half their normal speed. Assuming the context switching adds no overhead, \( p_2 \) is finished at time 6. Thereafter, \( p_1 \) is allowed to run to completion.

**Multilevel Priority**

The multilevel priority (ML) scheme uses a fixed set of priorities, 1 through \( n \), where 1 is the lowest and \( n \) is the highest priority. Every process is assigned an external priority value, \( e \), which is recorded in the process descriptor and determines the process’ current priority. The assignment of the priority may occur once, at the time the process is created, or it may be changed explicitly at runtime by authorized kernel calls. Thus, the priority function may be expressed as \( P = e \).

The ML discipline is typically preemptive in that a newly arriving process preempts the currently running one if its priority is greater than that of the current process. Within each priority queue, scheduling may be preemptive or nonpreemptive. In the first case, RR scheduling is applied within each priority level. That means, processes at the highest-priority level are cycled through the CPU at each quantum interrupt until they all terminate or block. Then the next priority level is serviced in the same RR manner, and so on.
With the nonpreemptive decision mode, FIFO is used within each level. That means, processes at the highest-priority level are started in the order of their arrival and allowed to run until they terminate or block. When the current priority level becomes empty, the next level is serviced in the same way.

**EXAMPLE: ML Algorithm**

Assume that $p_1$ (Fig. 5-2) runs at priority level $e_{p_1} = 15$, and $p_2$ runs at priority level $e_{p_2} = 14$. Then $p_1$ will run to completion before $p_2$ is started. If both priority levels were equal, i.e., $e_{p_1} = e_{p_2}$, the decision mode would determine the sequence of execution. With a preemptive decision mode, the two processes would run as under the RR policy. With a nonpreemptive mode, they would run as under the FIFO policy.

**Multilevel Feedback**

Similar to ML, the multilevel feedback (MLF) scheduling policy also provides $n$ different priority levels. However, processes do not remain at the same priority level but gradually migrate to lower levels as their attained service time increases. With each priority level $P$ we associate a time $t_P$, which is the maximum amount of processor time any process may receive at that level. If a process exceeds $t_P$, its priority is decreased to $P - 1$. One possible scheme is to choose a constant value, $T$, for the highest priority level and double that value for each lower priority level. That means, $t_P = T$ for $P = n$, and $t_P = 2 \times t_{P+1}$ for $1 \leq P < n$.

The organization of the MLF queues is shown in Figure 5-3. A newly arrived process or a process awakened from a blocking operation is queued at the highest-priority level $n$, where it must share the processor with all other processes at the same level. When it has received the maximum of $T$ units of CPU time, it is moved to the next lower-priority queue, $n - 1$, where it can consume up to $2T$ units of CPU time, and so on. When it reaches the lowest level 1, it remains there for a period of $2^{n-1}T$ time units. Exceeding this time is interpreted as an error, i.e., a runaway process. The CPU is always serving the highest-priority nonempty queue. Thus, a process at a level $P$ will receive service only when the queues at the levels $n$ through $P + 1$ are empty.

Similar to ML, the decision mode of MLF may be preemptive or nonpreemptive within each priority queue. In the first case, processes at each level are sharing the processor in a RR fashion. Usually, each of the times $t_P$ is a multiple of the basic quantum $q$; thus, a process at level $P$ will receive $t_P/q$ time quanta before it is moved to the next lower-priority queue. If the nonpreemptive mode is used, each level follows the FIFO discipline: The process at the head of the highest-priority nonempty queue $P$ is assigned a processor and it executes until it runs to completion or it exhausts the maximum time $t_P$, at which time it is moved to the next lower-priority queue. The next process at the same level $P$ then continues.

The priority function for the MLF scheduling discipline is a function of the attained service time $a$ and can be derived as follows. The priority of a process changes from $P$ to $P - 1$ whenever its total attained service time exceeds the sum of all times attained at levels $n$ through $P$. Assuming the scheme where the maximum time at each level is twice that of the previous level, we observe the following pattern: $a < T$ at priority level $n$, $a < T + 2T$ at priority level $n - 1$, $a < T + 2T + 4T$ at priority level $n - 2$, ...
and so on. In general, \( a < (2^{i+1} - 1)T \) at priority level \( n - i \). Thus, to determine the priority \( P = n - i \) of a process given its attained service time \( a \), we must find the smallest integer \( i \) such that the condition \( a < (2^{i+1} - 1)T \) holds. Solving this inequality for \( i \) yields the expression \( i = \lfloor \log_2(a/T + 1) \rfloor \). Thus, the priority function for MLF is \( P = n - i = n - \lfloor \log_2(a/T + 1) \rfloor \).

With this expression, \( P = n \) when \( a < T \). As \( a \) increases, \( P \) steps through the values \( n - 1 \) through 1. When \( a \) exceeds the value \( (2^n - 1)T \), the priority \( P \) becomes zero, indicating an error.

**EXAMPLE: MLF Algorithm**

Assume that the maximum time a process may spend at the highest-priority level is \( T = 2 \). Both processes \( p_1 \) and \( p_2 \) start at the highest-priority level, \( n \), where each is allowed to use 2 units of time. \( p_2 \) will finish at that level, whereas \( p_1 \) will migrate to level \( n - 1 \), where it is allowed to spend up to \( 2T = 4 \) time units. Thus, \( p_2 \) will terminate its task at that level.

**Rate Monotonic**

In real-time systems, processes and threads are frequently periodic in nature so that computations are repeated at fixed intervals. Typically, the deadline for each computation
Section 5.2  Scheduling Methods  159

is the beginning of the next period. Thus, if a process has a period \( d \), it is activated every \( d \) units of time, and its computation (total service time \( t \)) must be completed before the start of the next period, i.e., within \( d \) time units.

The rate monotonic (RM) method is a preemptive policy that dispatches according to period length; the shorter the period, \( d \), the higher the priority. The priority function has the form \( P = -d \). For example, a computation that is repeated every hour has a higher priority than one repeated every hour. The arbitration rule is either random or chronological.

**EXAMPLE:  RM Algorithm**

To illustrate the use of RM, assume that both \( p_1 \) and \( p_2 \) are periodic processes with the following respective periods: \( d_1 = 9 \) and \( d_2 = 8 \). Thus, the priority of \( p_2 \) is always higher than the priority of \( p_1 \). Consequently, at time 2, process \( p_2 \) will preempt the currently executing process \( p_1 \), which will resume its execution at time 4, i.e., when the current execution of \( p_2 \) terminates.

Earliest-Deadline-First

The earliest-deadline-first (EDF) method is a preemptive and dynamic scheme, used primarily for real-time applications. The highest priority is assigned to the process with the smallest remaining time until its deadline. As in the RM case above, we assume that all processes are periodic and that the deadline is equal to the end of the current period. The priority function can be derived as follows. If \( r \) is the time since the process first entered the system and \( d \) is its period, then \( r / d \) is the number of completed periods and the remainder of this division, \( r \%d \) (i.e., \( r \) modulo \( d \)) is the already expired fraction of the current period. The time remaining until the end of this period is then \( d - r \%d \). Thus, the priority function has the form \( P = -(d - r \%d) \). For an arbitration rule, either a random or a chronological policy is defined typically.

**EXAMPLE:  EDF Algorithm**

Assume again the same periods for the two processes, \( d_1 = 9 \) and \( d_2 = 8 \). At time 2, the priority of \( p_1 \) is \(- (d_1 - r_1 \%d_1) = -(9 - 2 \%9) = -(9 - 2) = -7\), and the priority of \( p_2 \) is \(- (d_2 - r_2 \%d_2) = -(8 - 0 \%8) = -(8 - 0) = -8\). Since \(-7 > -8\), \( p_1 \) will continue executing, whereas \( p_2 \) is delayed until time 5, when \( p_1 \) terminates.

Figure 5-4 summarizes the main characteristics of the scheduling methods in terms of the decision mode, the priority function, and the arbitration rule.

5.2.3  Comparison of Methods

The first three algorithms discussed above—FIFO, SJF, and SRT—have been developed primarily for batch processing systems. FIFO is the simplest to implement, since processes are maintained in the order of their arrival on a queue. On the other hand, SJF requires that processes are serviced in the order defined by their time requirement, implying either a more complex priority queue data structure or lengthy sorting and searching operations.
In the case of SRT, a similar implementation is necessary. In addition, preemption of the currently running process may be performed whenever a new process enters the system.

If the total service times are known (or can be predicted), SJF or SRT are generally preferable to FIFO, because both favor shorter computations over long-running ones. This is important for two reasons. First, users will not be satisfied unless they receive fast service for short tasks. (Supermarkets have long recognized this elementary fact by providing special checkout counters for customers with few items.) The second reason derives from a theoretical consideration: The order in which processes are executed has no effect on the total time required to complete all processes. The preferential treatment of shorter processes, however, reduces the average time a process spends in the system. This time is referred to as the average turnaround time and is the most important metric used to compare many scheduling algorithms.

The average turnaround time for a set of processes is defined as the value of the expression \( \frac{\sum_{i=1}^{n} r_i}{n} \), where \( r_i \) is the real time each process \( i \) spends in the system during its lifetime and \( n \) is the total number of active processes. Generally, average turnaround time is shortest using SJF and SRT. For example, if two processes, \( p_1 \) and \( p_2 \), have total service times \( t_1 \) and \( t_2 \), respectively, with \( t_1 < t_2 \), and they both arrive at the same time, then an SJF policy selects process \( p_1 \) to run first. Thus for \( p_1 \), \( r_1 = t_1 \). Process \( p_2 \) must wait for \( t_1 \) time units and then runs for \( t_2 \) time units. Thus, \( r_2 = t_1 + t_2 \); the average turnaround time is \( \frac{t_1 + t_2}{2} \). This is less than a policy that executes process \( p_1 \) first, resulting in the average turnaround time of \( \frac{t_2 + (t_1 + t_2)}{2} \).

**EXAMPLE: FIFO, SJF, and SRT**

Consider three batch processes, \( p_1, p_2, \) and \( p_3 \), with the following arrival times and total service times:

### Table: Scheduling algorithms

<table>
<thead>
<tr>
<th>Scheduling algorithm</th>
<th>Decision mode</th>
<th>Priority function</th>
<th>Arbitration rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO</td>
<td>Nonpreemptive</td>
<td>( r )</td>
<td>Random</td>
</tr>
<tr>
<td>SJF</td>
<td>Nonpreemptive</td>
<td>(-t)</td>
<td>Chronological or random</td>
</tr>
<tr>
<td>SRT</td>
<td>Preemptive</td>
<td>(-(t - a))</td>
<td>Chronological or random</td>
</tr>
<tr>
<td>RR</td>
<td>Preemptive</td>
<td>0</td>
<td>Cyclic</td>
</tr>
<tr>
<td>ML</td>
<td>Preemptive</td>
<td>( e )</td>
<td>Cyclic</td>
</tr>
<tr>
<td>MLF</td>
<td>Preemptive</td>
<td>( n - \lfloor lg(a/T + 1) \rfloor ) (same)</td>
<td>Chronological</td>
</tr>
<tr>
<td>RM</td>
<td>Preemptive</td>
<td>(-d)</td>
<td>Chronological or random</td>
</tr>
<tr>
<td>EDF</td>
<td>Preemptive</td>
<td>(-(d – r % d))</td>
<td>Chronological or random</td>
</tr>
</tbody>
</table>

- \( a = \) attained service time
- \( r = \) real time in system
- \( n = \) number of priority levels
- \( t = \) total service time
- \( T = \) maximum time at highest level
- \( d = \) period
- \( e = \) external priority

**FIGURE 5-4.** Characteristics of scheduling methods.
Section 5.2  Scheduling Methods   161

<table>
<thead>
<tr>
<th>arrival</th>
<th>service</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>$t$</td>
</tr>
<tr>
<td>$p_2$</td>
<td>$t$</td>
</tr>
<tr>
<td>$p_3$</td>
<td>$t+3$</td>
</tr>
</tbody>
</table>

Figure 5-5 shows a timing diagram for completing the three processes under FIFO, SJF, and SRT scheduling disciplines.

- **FIFO**: Since both $p_1$ and $p_2$ arrived at the same time, we assume that $p_1$ is started first. $p_2$ and $p_3$ then follow in the order of their arrival.

- **SJF**: $p_2$ has a shorter service time than $p_1$ and is started first. When it terminates, $p_1$ is the only process waiting and is started. Finally, $p_3$, which arrived in the meantime, is processed.

- **SRT**: $p_2$ and $p_1$ begin at the same respective times as under SJF. However, when $p_3$ arrives, its remaining time is only 1 unit of time whereas the remaining time of $p_1$ is 3. Thus $p_1$ is preempted and must wait for the completion of $p_3$.

Figure 5-6 shows the turnaround times for each process, computed as the sum of the waiting time and the total service time. The last column shows the average turnaround times.

**FIGURE 5-5.** Process completion under different scheduling disciplines.

<table>
<thead>
<tr>
<th></th>
<th>$p_1$</th>
<th>$p_2$</th>
<th>$p_3$</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO</td>
<td>0 + 4</td>
<td>4 + 2</td>
<td>3 + 1</td>
<td>14/3 = 4.66</td>
</tr>
<tr>
<td>SJF</td>
<td>2 + 4</td>
<td>0 + 2</td>
<td>3 + 1</td>
<td>12/3 = 4.00</td>
</tr>
<tr>
<td>SRT</td>
<td>3 + 4</td>
<td>0 + 2</td>
<td>0 + 1</td>
<td>10/3 = 3.33</td>
</tr>
</tbody>
</table>

**FIGURE 5-6.** Process turnaround times.
162 Chapter 5 Process and Thread Scheduling

turnaround time for all three processes. FIFO gives the largest turnaround time, whereas SRT produces the shortest.

In time-shared systems, where a number of users interact with the system in the form of a dialog, a scheduler should guarantee an acceptable response time for each user. This implies the use of a preemptive algorithm with preemption of processes at definite time intervals predetermined by the system. Of the methods presented above, only RR and MLF satisfy this requirement. RR is the simplest to implement, and therefore it is relied on heavily in many existing systems. It divides the CPU time evenly among all active processes according to the quantum size $q$, which becomes the most important parameter of this class of algorithms. Ideally, all processes may be viewed as running simultaneously at the speed $\text{cpu.speed}/n$, where $\text{cpu.speed}$ is the speed of the CPU and $n$ is the number of active processes sharing this CPU. If $q$ is large, process switching is less frequent, and overhead is reduced; response time will, of course, deteriorate. In the extreme case, where $q \to \infty$, processes are never preempted, and the RR algorithm turns into a FIFO discipline. At the other extreme, where $q \to 0$, the overhead associated with process switching dominates and no work is done.

CASE STUDY: QUANTUM SIZE IN WINDOWS 2000

Windows 2000. The quantum is controlled by an entry in the internal database, the registry. The first choice is between a short and a long quantum. A short quantum lasts two clock ticks (or around 20 ms on an Intel 486 processor) and is usually used by general-purpose computer systems. A long quantum lasts 12 clock ticks. This reduces the context-switching overhead; thus, the long quantum is the preferred choice for servers, where high throughput is more important than high responsiveness.

In addition to selecting the length, the quantum may be set as fixed for all threads, or it may be different for background and foreground processes. In the latter case, the quanta are selected from a table of possible combinations, which range between two and 12 clock ticks (in increments of 2).

Frequently, more sophisticated preemption schemes are needed that discriminate among different groups of processes based on priorities. The criteria for assigning priorities may be based on a variety of parameters, including external values; more often, they are based on measurements of process behavior rather than a priori declarations. The static ML discipline uses a fixed classification of all processes. Most of the dynamic schemes are designed to move interactive and I/O-bound processes to the top of the priority queues and to let CPU-bound processes drift to lower levels. The MLF policy accomplishes this objective by always placing new processes and those awakened as a result of I/O completion into the highest-priority queue. Processes with short computation phases between I/O operations will not exhaust the maximum times associated with the top levels and will tend to retain higher priority over processes with few I/O operations.
1. VAX/VMS. A popular system of an earlier generation, VAX/VMS, has an elegant scheduling scheme that combines the principles of several disciplines discussed earlier (DIGITAL 1982). It supports 32 priority levels divided into two groups of 16. As illustrated in Figure 5-7a, priorities 31-16, where 31 is the highest, are reserved for real-time processes. Other processes, referred to as regular,

FIGURE 5-7. Priority levels (a) VAX/VMS; and (b) Linux.
occupy priority levels 15-0. Processes are dispatched according to their current priority. This guarantees that real-time processes have precedence over regular processes; thus, a regular process will be scheduled only when no real-time process is awaiting execution.

The priority assigned to a real-time process is fixed for the duration of that process. Thus the top 15 priority levels are processed according to the ML scheme (without feedback). On the other hand, priorities of regular processes float in response to various events occurring in the system. A base priority is assigned to each such process when it is created; this specifies its minimum priority level. The current priority of the process then varies dynamically with its recent execution history according to the following rules. Each system event has an assigned priority increment that is characteristic of the event cause. For example, terminal read completion has a higher increment value than terminal write completion, which, in turn, has a higher value than disk I/O completion. When a process is awakened as a result of one of these events, the corresponding increment is added to its current priority value, and the process is enqueued at the appropriate level. When the process is preempted after receiving its fair share of CPU usage, its current priority is decremented by one, which places it into the next lower-priority queue. Thus the priority of a regular process fluctuates between its base priority and the value 15, which is the maximum priority for any regular process.

Preemption of a currently running process from the CPU may be caused by a number of events. A real-time process is preempted only when it blocks itself, for example, by issuing an I/O instruction or when a higher-priority process arrives or is awakened; otherwise, it is permitted to run to completion. A regular process is preempted for the same reasons. In addition, it also is preempted when it exceeds a quantum assigned for the current priority level. When this occurs, the process is stopped and placed into the next lower-priority queue. Thus, regular processes are prevented from monopolizing the CPU.

Note that the scheduling algorithm for regular processes (levels 15-0) is a variant of the MLF discipline. The two main distinctions are:

(a) Under MLF, processes always start at the highest-priority level and, if not blocked, propagate toward the lowest level. The VMS scheduler restricts the priority range within which a process may float. The lowest level is given by the process base priority, whereas the highest level depends on the type of event that caused the reactivation of the process.

(b) A quantum is associated with a process rather than with a priority level, which permits the scheduler to discriminate among individual processes.

2. Windows 2000. The basic scheduling principles of Windows 2000 are very similar to those of the VAX/VMS. The main difference is that VAX/VMS schedules
Section 5.2  Scheduling Methods  165

processes, whereas Windows 2000 schedules threads within processes. A base priority is assigned to each process at creation. Threads within each process inherit this base priority as the default. Then, whenever a thread wakes up, it gets a boost according to the event it was waiting for. For example, a thread waiting for disk I/O will get a boost of 1, whereas a thread waiting for a mouse or keyboard interrupt gets a boost of 6. Thus, a thread enters the system at the current priority, which is equal to the thread’s base priority plus the current boost. As it continues executing, its current priority is decreased by one each time it exhausts the quantum until it again reaches the base priority. As in the case of VAX/VMS, real-time threads have a static priority; no boosts are added.

3. Minix. The Minix OS is a small version of UNIX and predecessor of Linux that is user-compatible with UNIX but with a different internal design (Tannenbaum and Woodhull 1997). The scheduler employs a ML discipline with three priority levels, called task, server, and user levels. A queue of ready processes is associated with each level.

The highest-priority processes are kernel processes at the task level. These consist primarily of I/O driver processes, for example, for the disk, clock, and Ethernet, and a system task that interfaces with servers. The server processes are assigned the middle priority. These implement higher-level OS functions such as memory, file, and network management. User processes have the lowest priority.

At the task and server levels, processes are run nonpreemptively until they block or terminate. User processes are scheduled using RR with a 100-ms quantum. If the three queues are all empty, an idle process is dispatched.

4. Linux. Linux employs a very elaborate scheduling scheme. One of its distinguishing features is that it does not maintain processes or threads sorted in a special priority queue. Instead, it orders them according to a dynamically computed value, referred to as the goodness of a thread. Although no explicit priority data structure exists, it is still convenient to visualize the threads arranged in a list sorted by their goodness. Figure 5-7b shows such a structure. Note that the goodness of a thread is similar to what other systems call current priority; it determines the importance of a thread relative to others.

The threads in Linux are divided into two basic classes, real-time and regular. Each real-time thread has a static priority assigned to it at the time of creation. This is an integer recorded in its PCB, which is added to the base value of 1000 to derive the thread’s goodness. This guarantees that the goodness of any real-time thread is always higher than that of a regular thread.

Real-time threads are subdivided further into FIFO and RR threads. This information also is recorded in the PCB of each thread. A FIFO thread always runs until completion. A RR thread is preempted when its current quantum expires but its goodness does not change. Thus, RR threads with the same goodness share the CPU in a RR fashion.

The goodness of a regular thread fluctuates between a base value and a current value. This is similar to the dynamic priorities of the VAX/VMS and Windows
2000. However, the base and current goodness values are directly tied to the quantum. That means, a quantum is not a fixed amount of CPU time associated with a given priority level. Instead, each thread has its own variable-length quantum, which, similar to a countdown timer, is decremented as the process executes. Specifically, the following rules apply:

- Each thread is assigned a **base quantum**, \( b_q \), at the time of creation. This value (also referred to as the thread’s priority) may be modified by an explicit call to the function \( \text{nice()} \) but otherwise remains constant.
- Each thread has a counter that keeps track of its **remaining quantum**, \( r_q \).
- The current **goodness**, \( g \), of the thread is computed using the following two rules:
  
  \[
  \text{if } (r_q == 0) \quad g = 0; \]
  \[
  \text{if } (r_q > 0) \quad g = b_q + r_q. \]

As a result, threads that have exhausted their current quantum are not permitted to run. The remaining threads are ordered by their goodness, which considers both their base quantum (base priority) and their remaining quantum.

Another distinguishing feature of the Linux scheduler is that it divides time into periods called **epochs**. An epoch ends when no threads are able to run. This happens when all threads have either exhausted their current quantum or are blocked. At this point, the scheduler resets the quanta of all threads, which starts the next epoch. The quanta are reset according to the following formula:

\[
 r_q = b_q + \left(\frac{r_q}{2}\right) \]

That means, each thread gets its base quantum plus half of its remaining quantum from the previous epoch. This naturally favors I/O-bound threads. A CPU-bound thread will have exhausted its quantum and thus only gets its base quantum for the next epoch. On the other hand, an I/O-bound thread gets to keep half of its unused quantum, in addition to getting a new base quantum.

Many processes, especially those that monitor or control signals from an external environment, must meet deterministic timing constraints, particularly deadlines. For such **real-time** processes, it is often important to know **before** their execution whether or not deadlines will be met. An assignment of processes to processors, called a **schedule**, is said to be **feasible** if all deadlines are satisfied. A scheduling method is defined as **optimal** if the schedule resulting from the use of the method meets all deadlines whenever a feasible schedule exists. In other words, an optimal algorithm produces a feasible schedule if one exists. (A more detailed discussion of real-time scheduling appears in [Shaw 2001]).

Each periodic process uses a fraction of the CPU. Specifically, a process \( p_i \) will use \( t_i/d_i \) of the CPU time, where \( d_i \) is the period length and \( t_i \) is the total service time
for process \( i \) for one period. For \( n \) processes, we define the overall CPU utilization as:

\[
U = \sum_{i=1}^{n} \frac{t_i}{d_i}
\]

Then scheduling is feasible using EDF as long as \( U \leq 1 \). This is the best that any algorithm can do, since it is impossible to produce a schedule if \( U > 1 \). With \( U = 1 \) the CPU is fully saturated. With RM, scheduling is feasible whenever \( U \) is less than approximately 0.7. The actual value depends on the particular set of processes. Furthermore, the condition is sufficient but not necessary; i.e., feasible schedules may still be possible for \( U > 0.7 \), but RM is not guaranteed to always find them.

**EXAMPLE: RM vs. EDF**

Let process \( p_1 \) have a period \( d_1 = 4 \) and service time \( t_1 = 1 \), and process \( p_2 \) have a period \( d_2 = 5 \) with service time \( t_2 = 3 \). Figure 5-8a shows the periods and service times for the two processes for the first 24 time units. On a uniprocessor, the two processes cannot run in parallel. The sequential schedule produced by RM is shown in Figure 5-8b. \( p_1 \) has a shorter period and always has a higher priority than \( p_2 \). Note that \( p_1 \) preempts \( p_2 \) whenever its new period begins (at times 12 and 16 in this example). The schedule meets all deadlines for both processes.

Figure 5-8c shows the schedule produced by EDF. For the first 12 time units the two schedules are identical, because \( p_1 \)'s remaining time until the next deadline is shorter than \( p_2 \)'s remaining time. At time 12, the situation changes. \( p_1 \)'s remaining time until the next deadline is 4; \( p_2 \)'s remaining time until the next deadline is 3. According to the priority function, \( p_1 \)'s priority is \(-4\) and \( p_2 \)'s priority is \(-3\); thus, \( p_2 \) continues, whereas \( p_1 \) must wait until time 13. An analogous situation occurs at time 16, where \( p_1 \) does not preempt \( p_2 \) but waits until time 18.

To illustrate that EDF always produces feasible schedules and RM does not, consider the same processes \( p_1 \) and \( p_2 \) but assume that \( p_1 \)'s service time is extended to \( t_1 = 1.5 \). The utilization of the system is \( U = t_1/d_1 + t_2/d_2 = 1.5/4 + 3/5 = 0.975 \). This is less than 1, and the processes are in principle schedulable. Figure 5-9 shows the two schedules. RM fails at time 5 where \( p_2 \) has not been able to complete its computation.
5.3 PRIORITY INVERSION

Most scheduling mechanisms in OSs are preemptive and priority-based, or can be transformed into or viewed as such. When employing these systems in realistic environments that include CSs and other resource interactions, a common problem occurs that conflicts with the intent of priorities: higher-priority processes or threads can be delayed or blocked, often unnecessarily, by lower-priority ones. This situation, called the priority inversion problem, was first publicized with respect to the Pilot OS and Mesa language (Lampson and Redell 1980), and later thoroughly investigated for real-time OSs (Sha, Rajkumar and Lehocsky 1990).

The problem is best illustrated through a standard example. Consider three processes, $p_1$, $p_2$, and $p_3$, such that $p_1$ has the highest priority, $p_3$ the lowest, and $p_2$ has priority between the other two. Suppose that $p_1$ and $p_3$ interact through CSs locked by a common semaphore $mutex$, and $p_2$ is totally independent of $p_1$ and $p_3$. The code skeletons follow.

```
p1: . . . P(mutex); CS_1; V(mutex); . . .
p2: . . . program_2; . . . /* independent of p1 and p3 */
p2: . . . P(mutex); CS_3; V(mutex); . . .
```

One possible execution history for a single processor is illustrated in Figure 5-10. $p_3$ starts first and enters its CS, $CS_3$. At time $a$, $p_2$ is activated and preempts $p_3$ because it has a higher priority. Process $p_1$ becomes ready and preempts $p_2$ at point $b$. When $p_1$ attempts to enter $CS_1$ at $c$, it blocks on $P(mutex)$ because the semaphore is still held by the lowest-priority process $p_3$. Process $p_2$ therefore continues its execution until it terminates at $d$. Then, $p_3$ finishes its CS, releases $mutex$ waking up $p_1$, and is therefore preempted by $p_1$ at time $e$.

This scenario is certainly logically correct; it does not violate the specification of the scheduler given in Section 5.1.2. But the behavior is counterintuitive because the highest-priority process is blocked, perhaps for an inordinately long time. The problem
is not that \( p_3 \) is blocking \( p_1 \); that is unavoidable since they both compete for the same resource, probably at unknown times. The problem is that the intermediate process \( p_2 \), which is totally independent of \( p_1 \) and \( p_3 \), is prolonging the blocked state of \( p_1 \). In fact, the blocking of \( p_1 \) by \( p_2 \) may in general last an unbounded and unpredictable amount of time. The highest-priority process is essentially blocked by an unrelated process of lower priority. Such a priority inversion is undesirable in general-purpose OSs and intolerable in real-time systems.

There are several solutions that avoid the inversion. One possibility is to make CSs nonpreemptable. This may be practical if CSs are short and few. However, it would not be satisfactory if high-priority processes often found themselves waiting for lower-priority processes, especially unrelated ones. A more practical solution is to execute CSs at the highest priority of any process that might use them. Using this policy in our example, process \( p_3 \) would execute \( CS_3 \) at \( p_1 \)'s priority, thus blocking \( p_1 \) for the time it takes to complete its CS. This simple technique, however, suffers from overkill and leads to other forms of priority inversion; higher-priority processes now cannot preempt lower-priority ones in their CSs, even when the higher ones are not currently trying to enter the CS. For example, \( p_1 \) might not try to enter \( CS_1 \) until well after \( p_3 \) has exited its CS, but \( p_1 \) still would not be able to preempt \( p_3 \) while \( p_3 \) is inside \( CS_3 \). Similarly, \( p_2 \) would not be able to preempt \( p_3 \) while \( p_3 \) is inside \( CS_3 \).

A practical set of methods that does not have the above disadvantages is based on dynamic priority inheritance (Sha, Rajkumar and Lehocsky 1990). In its purest form, priority inheritance supports the following protocol. Whenever a process \( p_1 \) attempts to enter a CS and is blocked by a lower-priority process \( p_3 \) that is already in a CS (locked by the same lock or semaphore), then \( p_3 \) inherits \( p_1 \)'s priority and keeps it until it leaves the CS, releasing the lock. The execution of Figure 5-10 is repeated in Figure 5-11, only this time with priority inheritance. At point \( c \) when process \( p_1 \) blocks, \( p_3 \) inherits \( p_1 \)'s high priority and resumes execution until it exits from \( CS_3 \). Priorities then revert to normal, and \( p_1 \) runs to completion at \( e \), followed by resumption of \( p_2 \).

The protocol works for any number of higher-priority processes becoming blocked on the same lock. The lock holder keeps inheriting the priority of the highest-priority blocked process. It also is applicable to nested CSs. Blocking times for the higher-priority processes are bounded by the CS execution times of lower-priority processes. The protocol can be implemented relatively efficiently as part of locking and unlocking.
operations, such as operations on mutual exclusion semaphores, since blocked processes are normally linked directly through a semaphore or lock data structure.

Priority inheritance methods are supported in a number of systems, and are compulsory requirements in real-time OSs. As examples, many of the real-time versions of UNIX conform to the IEEE POSIX standard and its real-time extensions.

1. **POSIX**. POSIX supports two mechanisms for preventing priority inversion. The simpler one allows the user to associate a ceiling attribute with a mutex semaphore. The ceiling is an integer, specified using the function `pthread_mutexattr_setprioceiling`. The corresponding protocol to use this ceiling is specified as `POSIX_THREAD_PRIO_PROTECT`. Then, when a thread locks this mutex, its priority is automatically raised to the value of the ceiling. Thus, all threads using this mutex will run at the same level and will not preemption each other.

The second mechanism is the actual priority inheritance, as described in this section. The user must specify the use of the protocol as `POSIX_THREAD_PRIO_INHERIT`. Then, no thread holding the mutex can be preempted by another thread with a priority lower than that of any thread waiting for the mutex.

2. **Windows 2000**. Windows 2000 addresses the priority inversion problem indirectly, through scheduling. A routine called the balance set manager scans all ready threads every second. If it finds a thread that has been idle for more than approximately 3 seconds, it boosts its priority to 15 (the highest non-real-time priority) and doubles its quantum. This prevents starvation of even the lowest-priority thread. Thus, no thread will be postponed indefinitely because of a possibly unrelated higher-priority thread; it will be able to complete its CS, which indirectly avoids the priority inversion problem.

5.4 MULTIPROCESSOR AND DISTRIBUTED SCHEDULING

We consider scheduling and dispatching issues for shared-memory multiprocessors and for tightly coupled distributed systems, such as LANs. There are two principal approaches to scheduling processes and threads on multicomputers:
Section 5.4  Multiprocessor and Distributed Scheduling  171

• Use a single scheduler. All CPUs are in the same resource pool, and any process can be allocated to any processor.

• Use multiple schedulers. The computers are divided into sets of separately scheduled machines, and each process is preallocated permanently to a particular set. Each set has its own scheduler.

In the previous sections, the first approach was assumed. Whenever a scheduling decision was to be made, a single mechanism considered all CPUs and all eligible processes or threads. Generally, this approach is most applicable to multiprocessor clusters, where all machines have the same features. One of the main goals is to evenly distribute the load over the machines—a technique called load balancing—for which many different algorithms have been developed. The second approach (using multiple schedulers), is often the choice for nonuniform multiprocessors, where different CPUs have different characteristics and functions. For example, there might be separate machines and associated schedulers for conventional I/O, filing, data acquisition, fast Fourier transforms, matrix computations, and other applications. Users and applications also have greater control over performance when there is support for more than one scheduler.

**CASE STUDIES: MULTIPROCESSOR SCHEDULING**

1. The Mach Scheduler. The Mach OS for multiprocessors employs the second approach (Black 1990). Each machine set has a global RL of schedulable threads. In addition, each CPU has a local ready queue for those threads permanently assigned to that CPU. Dedicating specific threads to a given processor may be necessary for UNIX compatibility or because the threads work only for a particular machine. Threads that handle I/O on devices attached to a particular CPU fit into the latter category. Dispatching occurs by selecting the highest priority thread from the local queue. If the local queue is empty, the highest-priority thread is dispatched from the global list. An idle thread associated with each CPU is executed when both lists are empty.

2. Windows 2000. For multiprocessor systems, Windows 2000 implements a form of scheduling called *affinity scheduling*. It maintains two integers associated with every thread: the *ideal processor* and the *last processor*. The ideal processor is chosen at random when a thread is created. The last processor is the one on which the thread ran last. The two numbers are used as follows:

   • When a thread becomes ready, it looks for a currently idle processors in the following order: 1) the ideal processor; 2) the last processor; 3) the processor currently running the scheduler; and 4) any other processor. If there is no idle processor, the scheduler tries to preempt a currently running thread, using the above order of desirability when examining the processors.

   • When a processor becomes idle (because a thread terminates or blocks), the scheduler looks for a thread to run on that processor. It makes the
CASE STUDIES: MULTIPROCESSOR SCHEDULING (continued)

selection using the following order: 1) the thread that last ran on the processor; and 2) the thread for which this is the ideal processor. If neither of these threads is ready, the scheduler selects a thread based on current priorities.

The main objective of affinity scheduling is to keep a thread running on the same processor as possible. This increases the chances that the cache still contains relevant data and improves performance.

Distributed systems normally have separate schedulers at each site (cluster), where either of the above two approaches are used at the local level, depending on the application and configuration. Processes are usually preallocated to clusters, rather than scheduled across the network using a single scheduler, for several reasons. First, communication is costly in time, and scheduling across a network inevitably requires much message traffic. A second reason is applications with shared address spaces as implemented by threads. Although distributed shared-memory and migrating threads are practical ideas, their realization is still slow compared with a conventional architecture. Hence, for performance reasons, threads that share memory are scheduled on the same multiprocessor set. Distributed systems also frequently have functional distribution, leading to different schedulers. Thus, in a typical client/server environment, different server nodes might exist for filing, email, printing, and other services, each using its own scheduler.

For real-time systems, neither RM nor EDF are optimal for multiprocessors. However, they are often satisfactory heuristics, in that they typically produce feasible schedules when they exist and when machines are not too heavily loaded. Thus, for distributed real-time systems, preallocation of processes to nodes is most common because of the complexity of trying to meet real-time constraints.

CONCEPTS, TERMS, AND ABBREVIATIONS

The following concepts have been introduced in this chapter. Test yourself by defining and discussing each keyword or phrase.

| Deadline | Priority ceiling |
| Dispatcher | Priority inheritance |
| Distributed systems scheduling | Priority inversion |
| Earliest-deadline-first (EDF) | Rate monotonic (RM) |
| Feasible schedule | Round-robin (RR) |
| First-in, first-out (FIFO) | Scheduler |
| Load balancing | Service time |
| Multilevel feedback (MLF) | Shortest-job-first (SJF) |
| Multiprocessor scheduling | Shortest-remaining-time (SRT) |
| Optimal schedule | Time quantum |
| Preemptive scheduling | |
EXERCISES

1. The scheduler of Section 5.1.2 is invoked as part of every Suspend, Activate, and Destroy operation on processes. For each of the three operations, determine how many times the function AllocateCPU() and the function Preempt() would be invoked, assuming that at the time the scheduler is called there are:
   (a) 3 free CPUs
   (b) 0 free CPUs

2. Simplify the scheduler of Section 5.1.2 for the following cases:
   (a) There is only a single CPU
   (b) Processes are never preempted; a process gives up the CPU only when it blocks on a request or terminates (assume that termination changes the process status to “terminated”)
   (c) Both restrictions (a) and (b) are combined

3. Consider the following set of five processes where arrival is the time the process became ready, t is the total service time, and e is the external priority.

<table>
<thead>
<tr>
<th>process</th>
<th>arrival</th>
<th>t</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>p0</td>
<td>0</td>
<td>80</td>
<td>9</td>
</tr>
<tr>
<td>p1</td>
<td>15</td>
<td>25</td>
<td>10</td>
</tr>
<tr>
<td>p2</td>
<td>15</td>
<td>15</td>
<td>9</td>
</tr>
<tr>
<td>p3</td>
<td>85</td>
<td>25</td>
<td>10</td>
</tr>
<tr>
<td>p4</td>
<td>90</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

   Assume that execution starts immediately at time 0 and there is no context switch overhead. For the following scheduling disciplines, draw a time diagram showing when each of the five processes executes. (In the case of a tie, assume that the process with the lower process number goes first.)
   (a) FIFO
   (b) SJF
   (c) SRT
   (d) RR (quantum = 1)
   (e) ML (with FIFO at each priority level)
   (f) MLF (with n = 5, T = 10, and FIFO at each priority level)

4. For each of the scheduling disciplines in the previous exercise, determine the average turnaround time for the five processes.

5. Assume that at time $t_0$, there are three processes $p0$, $p1$, and $p2$, in the system. Their arrival times, total service times ($t$), external priorities ($e$), and periods ($d$) are listed in the table below. Execution of $p0$ started immediately at its arrival. For each scheduling discipline, give the numerical priority (as defined by the appropriate priority function) of each of the three processes at time $t_0$.

<table>
<thead>
<tr>
<th>process</th>
<th>arrival</th>
<th>t</th>
<th>e</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>p0</td>
<td>$t_0 - 20$</td>
<td>50</td>
<td>0</td>
<td>160</td>
</tr>
<tr>
<td>p1</td>
<td>$t_0 - 10$</td>
<td>60</td>
<td>1</td>
<td>200</td>
</tr>
<tr>
<td>p2</td>
<td>$t_0$</td>
<td>20</td>
<td>2</td>
<td>150</td>
</tr>
</tbody>
</table>

   (a) FIFO
   (b) SJF
   (c) SRT
   (d) RR
   (e) ML
Chapter 5 Process and Thread Scheduling

6. Consider $n$ processes sharing the CPU in a RR fashion.
   (a) Assuming that each context switch takes $s$ milliseconds, determine the quantum size $q$ such that the overhead resulting from process switching is minimized but, at the same time, each process is guaranteed to get its turn at the CPU at least every $t$ seconds.
   (b) If $n = 100$, $t = 1$, and $s = 0.001$, what would be the size of $q$? What if $s$ increased to 0.01?

7. Consider a system using RR scheduling with a fixed quantum $q$. Every context switch takes $s$ milliseconds. Any given process runs for an average of $t$ milliseconds before it blocks or terminates.
   (a) Determine the fraction of CPU time that will be wasted because of context switching for each of the following cases:
      i. $t < q$
      ii. $t \gg q$ (i.e., $t$ is much greater than $q$)
      iii. $q$ approaches 0
   (b) Under what condition will the wasted fraction of CPU time be exactly 50%?

8. Consider a system with an MLF scheduling policy, where $n = 5$, $T = 2$, $t_P = 2 \times t_{P+1}$ for all $1 \leq P \leq n$, and FIFO is used within each priority level. At time 0, three processes $p_0$, $p_1$, and $p_2$ arrive in the highest-priority queue in that order. Their respective times until the next I/O request are 3, 8, and 5 time units. When a process reaches its I/O request, it remains blocked (not competing for the CPU) for 5 time units, after which it reenters the highest-priority queue. The required CPU times until the next I/O request are again 3, 8, and 5.
   Using a timing diagram, show for the first 40 time units of execution (a) which process will be running during each time unit and (b) at which priority level.

9. Repeat the previous exercise, assuming that $t_P = 2$ (i.e., constant) for all priority levels.

10. Derive an expression for the priority function of an MLF scheduling discipline where $t_P = T$ for all $1 \leq P \leq n$, i.e., the maximum time a process can spend at any priority level is constant.

11. Consider three processes, $p_0$, $p_1$, and $p_2$. The following tables show four different sets of values for the total service time ($t$) and period ($d$) of each process.

<table>
<thead>
<tr>
<th></th>
<th>$t$</th>
<th>$d$</th>
<th></th>
<th>$t$</th>
<th>$d$</th>
<th></th>
<th>$t$</th>
<th>$d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_0$</td>
<td>3</td>
<td>50</td>
<td>$p_0$</td>
<td>15</td>
<td>50</td>
<td>$p_0$</td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>$p_1$</td>
<td>70</td>
<td>1000</td>
<td>$p_1$</td>
<td>5</td>
<td>10</td>
<td>$p_1$</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>$p_2$</td>
<td>5</td>
<td>40</td>
<td>$p_2$</td>
<td>4</td>
<td>1</td>
<td>$p_2$</td>
<td>4</td>
<td>100</td>
</tr>
</tbody>
</table>

   (a) Determine for which of the four cases a feasible schedule is likely to be generated by:
      i. RM
      ii. EDF
   (b) For each case, show a timing diagram for the first 25 time units.

12. Consider four processes, $p_0$ through $p_3$. The diagram below shows the timing of each process when executed in isolation. That means, process $p_0$ arrives at time 0; after one time unit it executes a $P(s)$ operation ($s$ is initially 1); after 3 more time units it executes a $V(s)$ operation and
terminates. Processes $p_1$ and $p_3$ are similar. Process $p_2$ does not execute any $P$ or $V$ operations. Assume that the execution of $P$ and $V$ is instantaneous, i.e., takes 0 time. Any context switch is also instantaneous. The priorities of the four processes are as follows: \( \text{priority}(p_0) < \text{priority}(p_1) < \text{priority}(p_2) < \text{priority}(p_4) \). Determine the start and end time of each of the four processes when executing concurrently on a single-processor system.

(a) without priority inheritance
(b) with priority inheritance
(c) without priority inheritance but making all CSs noninterruptible

(Hint: Draw timing diagrams similar to those in Figures 5-7 and Figure 5-8.)

13. Assume two processes compete for a CS implemented using simple spinning locks (Section 4.5.1). Prior to entering the CS, the process executes the statement

\[
\text{do TS}(R, sb) \text{ while } (!R);
\]

After exiting the CS, it executes the statement:

\[
sb = 1;
\]

Show how this solution may lead to a deadlock under priority scheduling, whereas RR scheduling does not suffer from this problem.

14. Consider the following solution to the bounded-buffer problem (Section 2.4.3):

\[
\text{semaphore } e = n, f = 0, b = 1;
\]

\[
\text{cobegin}
\]

Producer: \( \text{while } (1) \{ \)

\[
\text{while } (\text{record_not_available}) ; \quad P(e); P(b); \text{Add_record_to_buffer}; V(b); V(f);
\]

\[
//
\]

Consumer: \( \text{while } (1) \{ \)

\[
P(f); P(b); \text{Take_record_from_buffer}; V(b); V(e);
\]

\[
\text{while } (\text{record_not_processed}) ;
\]

\[
\text{coend}
\]

Note that the producer has a busy-wait loop to wait for the next record to become available. Similarly, the consumer has a busy-wait loop to wait for the current record to be processed. Simplify this code under the assumption that it will be running on a single-processor system with nonpreemptive scheduling. A process runs until it
voluntarily gives up the CPU by executing the function \textit{yield()}. (Hint: There is no need for any semaphores; instead, the consumer and producer processes yield the CPU to each other so that both make progress while preserving the integrity of the buffer.)