Chapter 3

PROCESS SCHEDULING

3.1 Basic Concepts

The objective of multiprogramming is to have some process running at all times to maximize CPU utilization.

Scheduling 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.

3.1.1 CPU-I/O Burst Cycle

The success of CPU scheduling depends on the following observed property of processes: Process execution consists of a cycle of CPU execution and I/O wait. Processes alternate back and forth 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 last CPU burst will end with a system request to terminate execution, rather than with another I/O burst (Figure 3.*1).*

The duration of these CPU bursts have been extensively measured. Although they vary greatly from process to process and computer to computer, they tend to have a frequency curve similar to that shown in Figure 3.2.

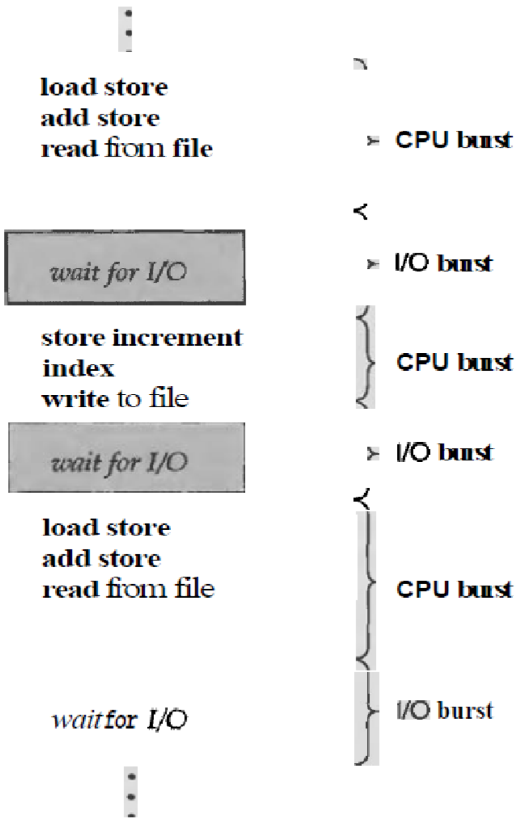


Figure 3.1 Alternating sequence of CPU and *I/O* bursts.

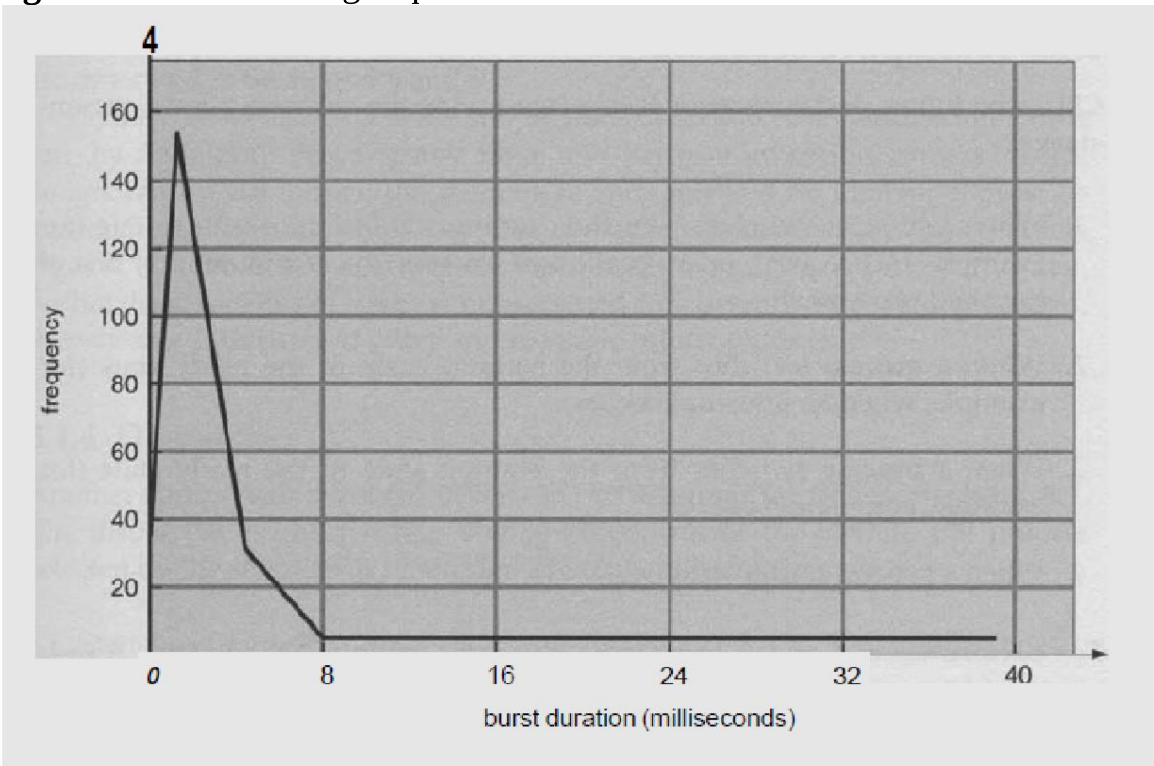


Figure 3.2 Histogram of CPU-burst times.

The curve is generally characterized as exponential or hyperexponential. There is a large number of short CPU bursts, and there is a small number of long CPU bursts. An I/O-bound program would typically have many very short CPU bursts. A CPU-bound program might have a few very long CPU bursts. This distribution can be important in the selection of an appropriate CPU scheduling algorithm.

3.1.2 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 from among the processes in memory that are ready to execute, and allocates the CPU to one of them.

3.1.3 Preemptive Scheduling

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

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

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

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

4. When a process terminates

For circumstances 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 circumstances 2 and 3.

When scheduling takes place only under circumstances 1 and 4, we say the scheduling scheme is ***nonpreemptive;*** otherwise, the scheduling scheme 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 is used by the Microsoft Windows environment.

Preemptive scheduling incurs a cost. Consider the case of two processes sharing data. One may be in the midst of updating the data when it is preempted and the second process is run. The second process may try to read the data, which are currently in an inconsistent state. New mechanisms thus are needed to coordinate access to shared data.

Preemption also has an effect on 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, 1/0 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. Some 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.

3.1.4 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:

1. Switching context

2. Switching to user mode

3. Jumping to the proper location in the user program to restart that program

The dispatcher should be as fast as possible, given that 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.***

3.2 Scheduling Criteria

Different CPU scheduling algorithms have different properties and may favour 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 the determination of the best algorithm. Criteria that are used include the following:

1. **CPU utilization:**

We want to keep the CPU as busy as possible. CPU utilization may 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).

2. 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, throughput might be 10 processes per second..

3. 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.

4. 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.

5. 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 amount of time it takes to start responding, but not the time that it takes to output that 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, there are circumstances when 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.

3.3 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 these algorithms.

3.3.1 First-Come, First-Served Scheduling

By far the simplest CPU scheduling algorithm is the ***first-come, first-served scheduling*** **(FCFS)** 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 time given in milliseconds:

Process	**Burst Time**
PI	**24**
P2	**3**
P3	**3**

If the processes arrive in the order PI, P2, P3, and are served in FCFS order, we get the result shown in the following ***Gantt chart:***

The waiting time is *0* milliseconds for process P1, **24** milliseconds for process P2, and **27** milliseconds for process P3. Thus, the average waiting time is *(0* + **24** + **27)/3** = **17** milliseconds. If the processes arrive in the order P2, P3, PP 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 a FCFS policy is generally not minimal, and may vary substantially if the process 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 the CPU and hold it. During this time, all the other processes will finish their I/O and 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 very short CPU bursts, execute quickly and move back to the 1/0 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 non preemptive. 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 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.

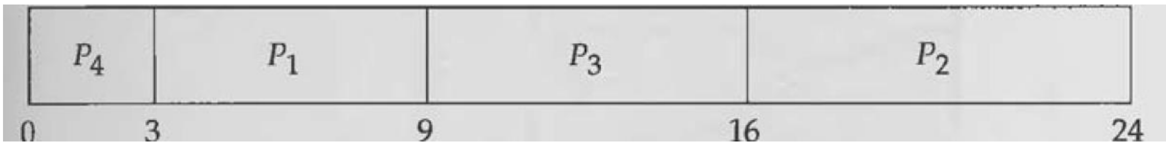
3.3.2 Shortest-Job-First Scheduling

A different approach to CPU scheduling is the shortest-job-first *(SJF)* algorithm. This algorithm associates with each process the length of the latter's next CPU burst. When the CPU is available, it is assigned to the process that has smallest next CPU burst. If two processes have the same length next CPU burst, FCFS scheduling is used to break the tie. Note that a more appropriate term would be the shortest next *CPU* burst, because the scheduling is done by examining 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 refer to this type of scheduling discipline as SJF.

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

Process	Burst Time
P1	**6**
P2	**8**
P3	**7**
P4	**3**

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



The waiting time is 3 milliseconds for process PI, 16 milliseconds for process ***fir*** 9 milliseconds for process P3, and 0 milliseconds for process P4. Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds. If we were using the FCFS scheduling scheme, then 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. By moving a short process before a long one the waiting time of the short process decreases more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.

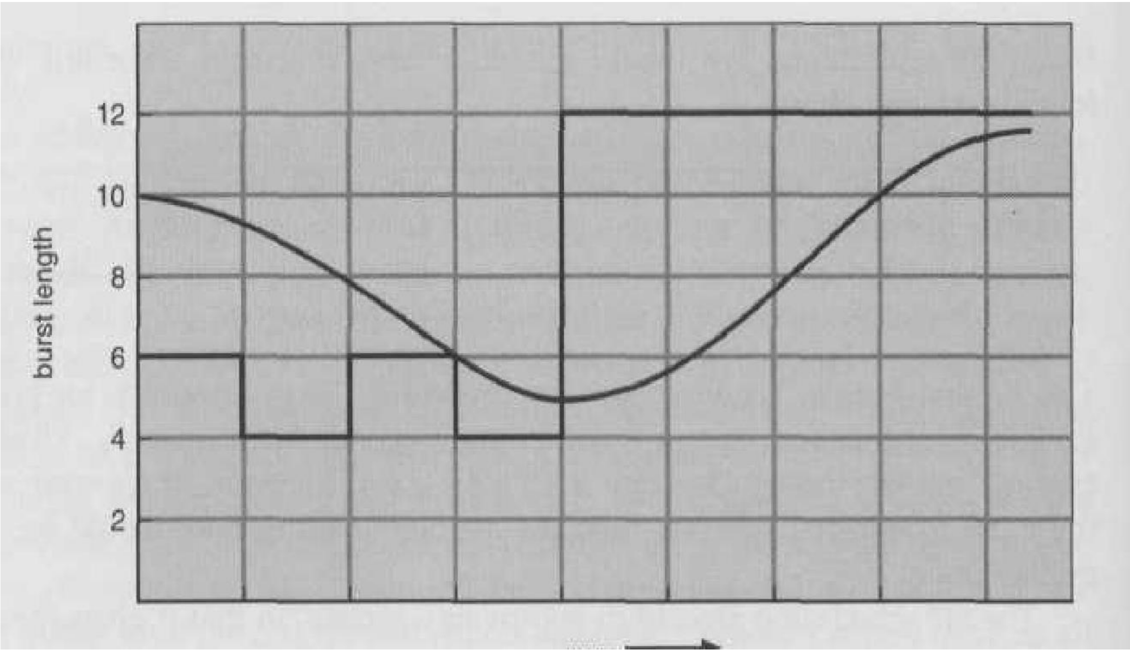
The real difficulty with the SJF algorithm is knowing the length of the next CPU request. For long-term (job) scheduling in a batch system, we can use as the length the process time limit that a user specifies when he submits the job. Thus, users are motivated to estimate the process time limit accurately, 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.

The next CPU burst is generally predicted as an exponential average of the measured lengths of previous CPU bursts. Let t, be the length of the nth CPU burst, and let Tn+1 be our predicted value for the next CPU burst. Then, for a, 0 < a *5* 1, define.

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

This formula defines an exponential average. The value of t, contains our most recent information; Tn stores the past history. The parameter a controls the relative weight of recent and past history in our prediction. If a = 0, then Tn+1 = ***Tn,*** and recent history has no effect (current conditions are assumed to be transient); if a = 1, then Tn+1 = tn, and only the most recent CPU burst matters (history is assumed to be old and irrelevant). More commonly, α = 1/2, so recent.



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

Figure 3.3 Prediction of the length of the next CPU burst.

History and past history are equally weighted. Figure 3.3 shows an exponential average with *a* = 1*II.* The initial *TQ* can be defined as a constant or as an overall system average.

To understand the behaviour of the exponential average, we can expand the formula for T_{n+1} by substituting for T_n, to find

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

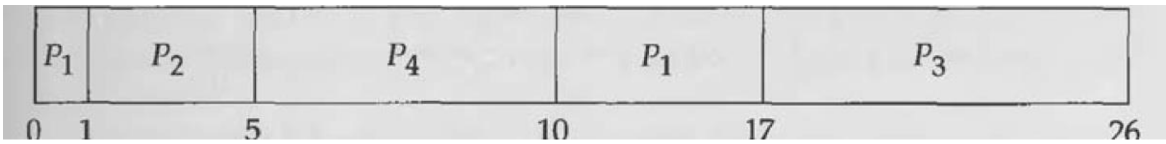
Since both *a* and (1 - *a)* are less than or equal to 1, each successive term has less weight than its predecessor.

The SJF algorithm may be either *preemptive* or *nonpreemptive*. The choice arises when a new process arrives at the ready queue while a previous process is executing. The new process may have a shorter next CPU burst 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 time given in milliseconds:

Process Arrival	Time	Burst Time
P1	0	8
P2	**1**	4
p3	2	9
p4	3	4

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 Pl is started at time *0*, since it is the only process in the queue. Process P2 arrives at time 1. The remaining time for process P1 (7 milliseconds) is larger than the time required by process P2 (4 milliseconds), so process PI is preempted, and process P2 is sche duled. The average waiting time for this example is ((10 - 1) + (1 - 1) + (17 - 2) + (5 - 3))/4 = 26/4 = 6.5 milliseconds.
A non preemptive SJF scheduling would result in an average waiting time of 7.75 Milliseconds.

3.3.3 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.

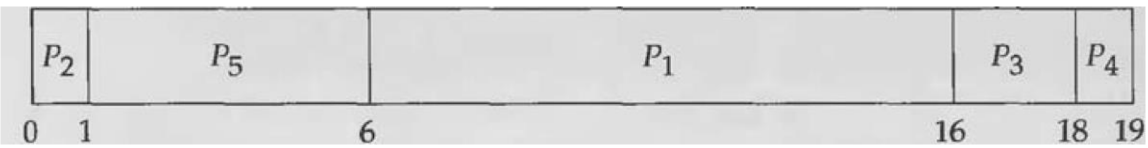
An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.

Note that we discuss scheduling in terms of ***high*** priority and ***low*** priority. Priorities are generally some fixed range of numbers, such as 0 to 7, or 0 to 4095. 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 PI, P2, ..., P5, with the length of the CPU-burst time 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

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 that are external to 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 lacking the CPU can be considered blocked, waiting for the CPU. A priority scheduling algorithm can leave some low-priority processes waiting indefinitely for the CPU. 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.

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 0 (low) to 127 (high), we could increment the priority of a waiting process by 1 every 15 minutes. Eventually, even a process with an initial priority of 0 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 0 process to age to a priority 127 process.

3.3.4 Round-Robin Scheduling

The round-robin (RR) scheduling algorithm is designed especially for timesharing 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.

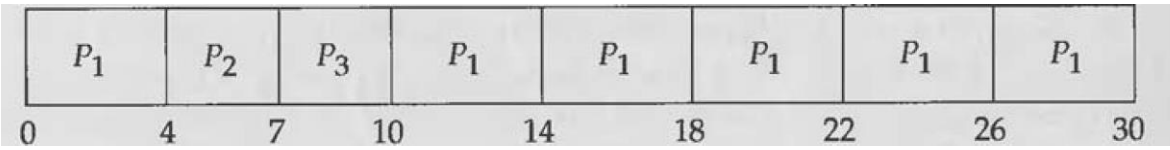
One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of the currently running process is 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, however, is often quite long. Consider the following set of processes that arrive at time 0, with the length of the CPU-burst time given in milliseconds:

Process	Burst Time
P_1	24
4	3
P_3	3

If we use a time quantum of 4 milliseconds, then process **PI** 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 **P2.** Since process **P2** does not need 4 milliseconds, it quits before its time quantum expires. The CPU is then given to the next process, process **P3.**

Once each process has received 1 time quantum, the CPU is returned to process **PI** 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. If a process' CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. The **RR** scheduling algorithm is preemptive.

If there are n processes in the ready queue and the time quantum is q, then each process gets l/n of the CPU time in chunks of at most q time units. Each process must wait no longer than (n - 1) x q time units until its next time quantum.

For example, if there are five processes, with a time quantum of 20 milliseconds, then 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 very large (infinite), the **RR** policy is the same as the FCFS policy. If the time quantum is very small (say 1 microsecond), the **RR** approach is called processor sharing, and appears (in theory) to the users as though each of n processes has its own processor running at l/n the speed of the real processor. This approach was used in Control Data Corporation (CDC) hardware to implement 10 peripheral processors with only one set of hardware and 10 sets of registers. The hardware executes one instruction for one set of registers, then goes on to the next. This cycle continues, resulting in 10 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 a single processor would have been.)

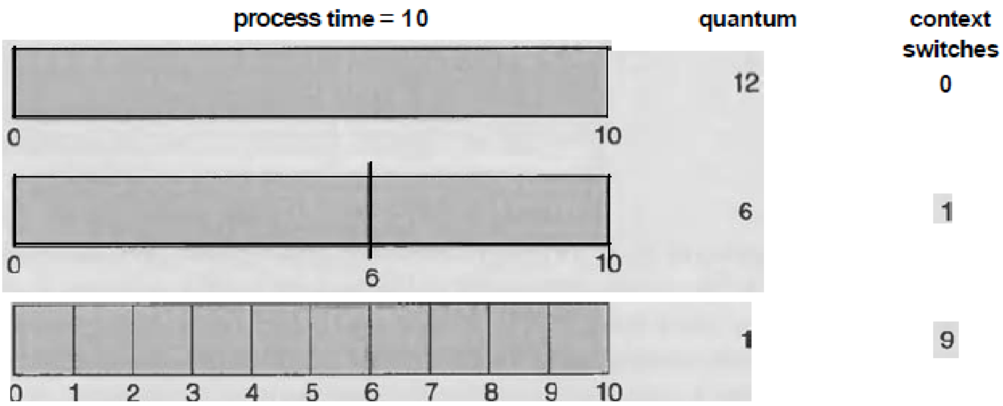
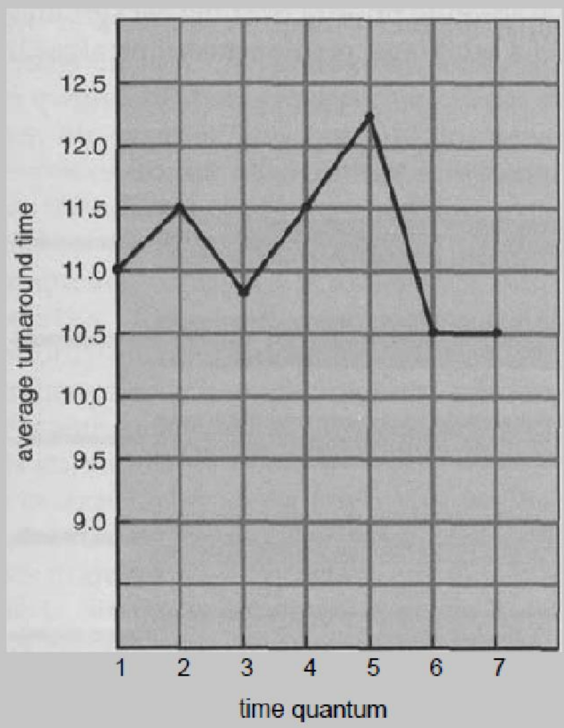


Figure 3.4 Showing how a smaller time quantum increases context switches.

In software, however, 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 3.4).

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 switch.

Turnaround time also depends on the size of the time quantum. As we can see from Figure 3.5, 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 will be required.



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Figure 3.*5* Showing how turnaround time varies with the time quantum.

On the other hand, 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.

3.3.5 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 common division is made between *foreground* (interactive) processes and *background* (batch) processes. These two types of processes have different response-time requirements, and so might 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 3.6). 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 between the queues, which is commonly implemented as a fixed-priority preemptive scheduling. For example, the foreground queue may have absolute priority over the background queue. Let us look at an example of a multilevel queue scheduling algorithm with five queues:

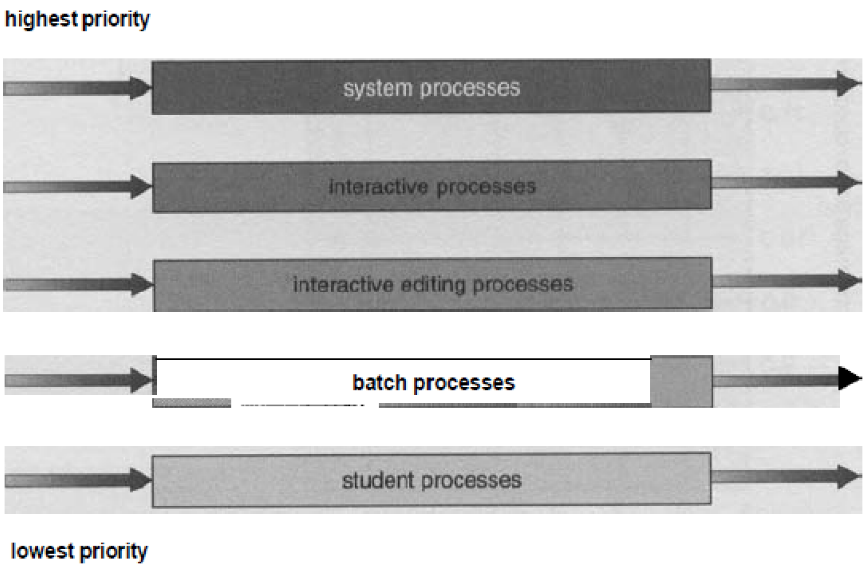


Figure 3.6 Multilevel queue scheduling.

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

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 between the queues. Each queue gets a certain portion of the CPU time, which it can then schedule among the various processes in its queue.

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 in a FCFS manner.

3.3.6 Multilevel Feedback Queue Scheduling

Normally, in a multilevel queue-scheduling algorithm, processes are permanently assigned to a queue on entry to the system. Processes do not move between queues. 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 is inflexible.

Multilevel feedback queue scheduling, however, allows a process to move between queues. The idea is to separate processes with different CPU-burst characteristics. 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. Similarly, 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 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,

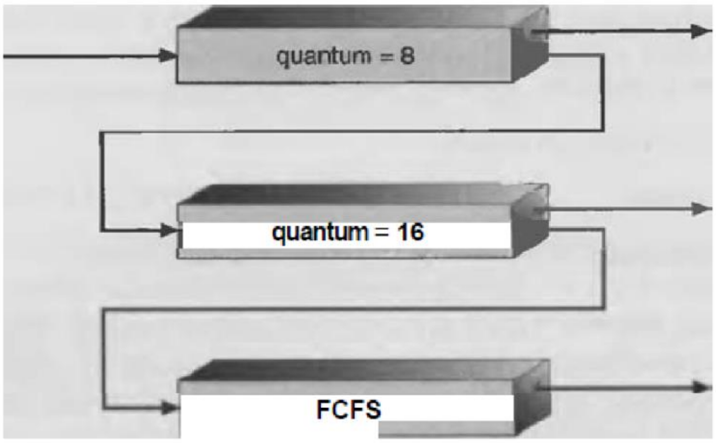


Figure 3.7 Multilevel feedback queues.

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, 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:

1. The number of queues
2. The scheduling algorithm for each queue
3. The method used to determine when to upgrade a process to a higher priority queue
4. The method used to determine when to demote a process to a lower-priority queue
5. 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 also requires some means of selecting values for all the parameters to define the best scheduler. Although a multilevel feedback queue is the most general scheme, it is also the most complex.

3.4 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, the scheduling problem is correspondingly more complex. Many possibilities have been tried, and, as we saw with single-processor CPU scheduling, there is no one best solution. In the following, we discuss briefly some of the issues concerning multiprocessor scheduling (A complete coverage is beyond the scope of this text.) We concentrate on systems where the processors are identical (homogeneous) in terms of their functionality; Any available processor can then be used to run any processes in the queue. only programs compiled for a given processor's instruction set could be run on that processor.

Even within homogeneous multiprocessor, there are sometimes limitations on scheduling. Consider a system with an I/O device attached to a private bus of one processor. Processes wishing to use that device must be scheduled to run on that processor, otherwise the device would not be available.
If several identical processors are available, then load sharing can occur. It would be possible to provide a separate queue for each processor. In this case, however, one processor could be idle, with an empty queue, while another processor was very busy. To prevent this situation, we use a common ready queue. All processes go into one queue and are scheduled onto any available processor.

In such a scheme, one of two scheduling approaches may be used. In one approach, each processor is self-scheduling. Each processor examines the common ready queue and selects a process to execute. If we have multiple processors trying to access and update a common data structure, each processor must be programmed very carefully.

We must ensure that two processors do not choose the same process, an d that processes are not lost from the queue. The other approach avoids this problem by appointing one processor as scheduler for the other processors, thus creating a master-slave structure.

Some systems carry this structure one step further, by having all scheduling decisions, I/O processing, and other system activities handled by one single processor - the master server. The other processors only execute user code.

This asymmetric multiprocessing is far simpler than symmetric multiprocessing, because only one processor accesses the system data structures, alleviating the need for data sharing.

3.4 Real-Time Scheduling

we gave an overview of real-time operating systems and discussed their growing importance. Here, we continue the discussion by describe ing the scheduling facility needed to support real-time computing within a general-purpose computer system.

Real-time computing is divided into two types. Hard real-time systems are required to complete a critical task within a guaranteed amount of time. Generally a process is submitted along with a statement of the amount of time in which it needs to complete or perform I/O. The scheduler then either admits the process, guaranteeing that the process will complete on time, or rejects the request as impossible. This is known as resource reservation. Such a guarantee requires that the scheduler know exactly how long each type of operating-system function takes to perform, and therefore each operation must be guaranteed to take a maximum amount of time. Such a guarantee is impossible in a system with secondary storage or virtual memory, as we shall show in the next few chapters, because these subsystems cause unavoidable and unforeseeable variation in the amount of time to execute a particular process.

Therefore, hard real-time systems are composed of special-purpose software running on hardware dedicated to their critical process, and lack the full functionality of modern computers and operating systems. Soft real-time computing is less restrictive. It requires that critical processes receive priority over less fortunate ones. Although adding soft real-time functionality to a time-sharing system may cause an unfair allocation of resources and may result in longer delays, or even starvation, for some processes, it is at least possible to ac hi eve. The result is a general-purpose system that can also support multimedia, high-speed interactive graphics, and a variety of tasks that would not function acceptably in an environment that does not support soft real-time computing.

Implementing soft real-time functionality requires careful design of the scheduler and related aspects of the operating system. First, the system must have priority scheduling, and real-time processes must have the highest priority. The priority of real-time processes must not degrade over time, even though the priority of non-real-time processes may Second, the dispatch latency must be small. The smaller the latency, the faster a real-time process can start executing once it is runnable.

It is relatively simple to ensure that the former property holds. For example, we can disallow process aging on real-time processes, thereby guaranteeing that the priority of the various processes does not change. However, ensuring the latter property is much more involved. The problem is that many operating systems, including most versions of UNIX, are forced to wait for either a system call to complete or for an 1/0 block to take place before doing a context switch.

The dispatch latency in such systems can be long, since some system calls are complex and some I/O devices are slow.To keep dispatch latency low, we need to allow system calls to be preemptible.

There are several ways to achieve this goal. One is to insert pre-emption points in long-duration system calls, which checks to see whether a high-priority process needs to be run.

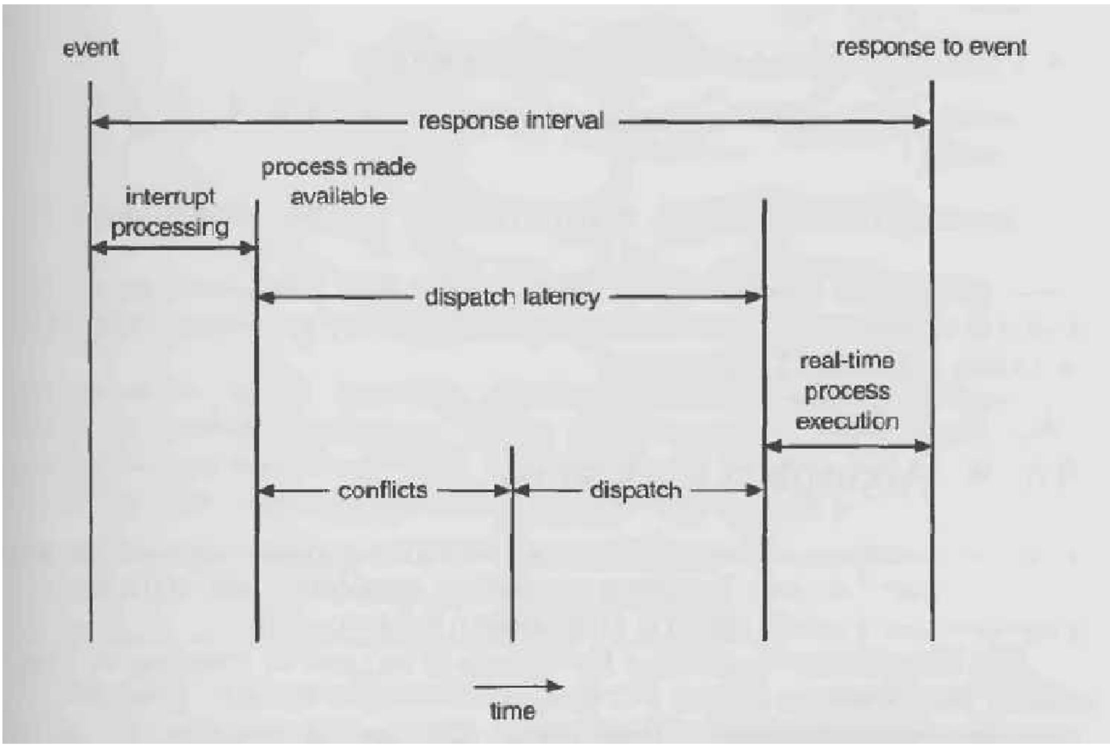


Figure 3.8 Dispatch latency

If so, a context switch takes place and, when the high priority process terminates, the interrupted process continues with the system calls. Preemption points can be placed at only "safe" locations in the kernel only where kernel data structures are not being modified. Even with pre-emption points dispatch latency can be large, because only a few preemption points can be practically added to a kernel.

Another method for dealing with preemption is to make the entire kernel preemptible. So that correct operation is ensured, all kernel data structures must be protected through the use of various synchronization mechanisms. With this method, the kernel can always be preemptible, because any kernel data being updated are protected from modification by the high-priority process. This is the method used in Solaris 2.

But what happens if the higher-priority process needs to read or modify kernel data that are currently being accessed by another, lower-priority process? The high-priority process would be waiting for a lower-priority one to finish. This situation is known as *priority inversion.* In fact, there could be a chain of processes, all accessing resources that the high-priority process needs. This problem can be solved via the *priority-inheritance protocol,* in which all these processes (the processes that are accessing resources that the high-priority process needs) inherit the high priority until they are done with the resource in question. When they are finished, their priority reverts to its natural value. In Figure 3.8, we show the makeup of dispatch latency. The *conflict phase* of dispatch latency has three components:

1. Preemption of any process running in the kernel
2. Low-priority processes releasing resources needed by the high-priority process
4. Context switching from the current process to the high-priority process

As an example, in Solaris 2, the dispatch latency with preemption disabled is over 100 milliseconds. However, the dispatch latency with preemption enabled is usually reduced to 2 milliseconds.