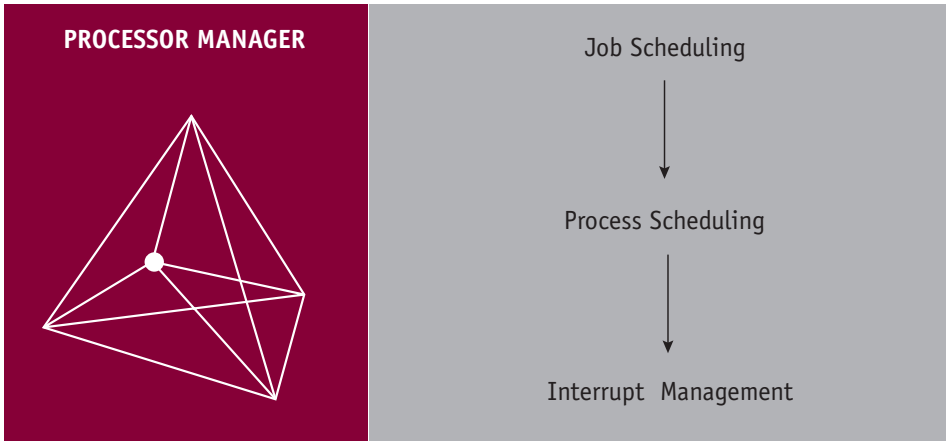
Chapter 4 Processor Management



"Nature acts by progress . . . It goes and returns, then advances further, then twice as much backward, then more forward than ever."

—Blaise Pascal *(1623–1662)*

Learning Objectives

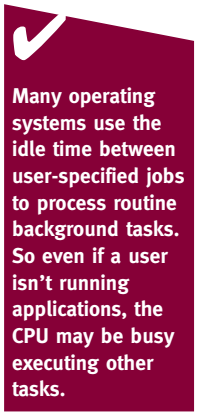
After completing this chapter, you should be able to describe:

- The difference between job scheduling and process scheduling, and how they relate
- The advantages and disadvantages of process scheduling algorithms that are preemptive versus those that are nonpreemptive
- The goals of process scheduling policies in single-core CPUs
- Six different process scheduling algorithms
- The role of internal interrupts and the tasks performed by the interrupt handler

The Processor Manager is responsible for allocating the processor to execute the incoming jobs, and the tasks of those jobs. In this chapter, we'll see how a Processor Manager manages a single CPU to do so.

Overview

In a simple system, one with a single user and one processor, the process is busy only when it is executing the user's jobs. However, when there are many users, such as in a multiprogramming environment, or when there are multiple processes competing to be run by a single CPU, the processor must be allocated to each job in a fair and efficient manner. This can be a complex task as we'll see in this chapter, which is devoted to single processor systems. Those with multiple processors are discussed in Chapter 6.



Many operating systems use the idle time between user-specified jobs to process routine background tasks. So even if a user isn't running applications, the CPU may be busy executing other tasks.

Before we begin, let's clearly define some terms. A **program** is an inactive unit, such as a file stored on a disk. A program is not a process. To an operating system, a program or job is a unit of work that has been submitted by the user.

On the other hand, a **process** is an active entity that requires a set of resources, including a processor and special registers, to perform its function. A process, also called a **task**, is a single instance of a program in execution.

As mentioned in Chapter 1, a **thread** is a portion of a process that can run independently. For example, if your system allows processes to have a single thread of control and you want to see a series of pictures on a friend's Web site, you can instruct the browser to establish one connection between the two sites and download one picture at a time. However, if your system allows processes to have multiple threads of control, then you can request several pictures at the same time and the browser will set up multiple connections and download several pictures at once.

The **processor**, also known as the CPU (for central processing unit), is the part of the machine that performs the calculations and executes the programs.

Multiprogramming requires that the processor be allocated to each job or to each process for a period of time and deallocated at an appropriate moment. If the processor is deallocated during a program's execution, it must be done in such a way that it can be restarted later as easily as possible. It's a delicate procedure. To demonstrate, let's look at an everyday example.

Here you are, confident you can put together a toy despite the warning that some assembly is required. Armed with the instructions and lots of patience, you embark on your task—to read the directions, collect the necessary tools, follow each step in turn, and turn out the finished product.

The first step is to join Part A to Part B with a 2-inch screw, and as you complete that task you check off Step 1. Inspired by your success, you move on to Step 2 and then Step 3. You've only just completed the third step when a neighbor is injured while working with a power tool and cries for help.

Quickly you check off Step 3 in the directions so you know where you left off, then you drop your tools and race to your neighbor's side. After all, someone's immediate

need is more important than your eventual success with the toy. Now you find yourself engaged in a very different task: following the instructions in a first-aid book and using bandages and antiseptic.

Once the injury has been successfully treated, you return to your previous job. As you pick up your tools, you refer to the instructions and see that you should begin with Step 4. You then continue with this project until it is finally completed.

In operating system terminology, you played the part of the *CPU* or *processor*. There were two *programs,* or *jobs*—one was the mission to assemble the toy and the second was to bandage the injury. When you were assembling the toy (Job A), each step you performed was a *process*. The call for help was an *interrupt;* and when you left the toy to treat your wounded friend, you left for a *higher priority program*. When you were interrupted, you performed a *context switch* when you marked Step 3 as the last completed instruction and put down your tools. Attending to the neighbor's injury became Job B. While you were executing the first-aid instructions, each of the steps you executed was again a *process*. And, of course, when each job was completed it was *finished* or terminated.

The Processor Manager would identify the series of events as follows:

`get the input for Job A`	(find the instructions in the box)
`identify resources`	(collect the necessary tools)
`execute the process`	(follow each step in turn)
`interrupt`	(neighbor calls)
`context switch to Job B`	(mark your place in the instructions)
`get the input for Job B`	(find your first-aid book)
`identify resources`	(collect the medical supplies)
`execute the process`	(follow each first-aid step)
`terminate Job B`	(return home)
`context switch to Job A`	(prepare to resume assembly)
`resume executing the interrupted process`	(follow remaining steps in turn)
`terminate Job A`	(turn out the finished toy)

As we've shown, a single processor can be shared by several jobs, or several processes—but if, and only if, the operating system has a scheduling policy, as well as a scheduling algorithm, to determine when to stop working on one job and proceed to another.

In this example, the scheduling algorithm was based on priority: you worked on the processes belonging to Job A until a higher priority job came along. Although this was a good algorithm in this case, a priority-based scheduling algorithm isn't always best, as we'll see later in this chapter.

About Multi-Core Technologies

A dual-core, quad-core, or other multi-core CPU has more than one processor (also called a core) on the computer chip. Multi-core engineering was driven by the problems caused by nano-sized transistors and their ultra-close placement on a computer chip. Although chips with millions of transistors that were very close together helped increase system performance dramatically, the close proximity of these transistors also increased current leakage and the amount of heat generated by the chip.

One solution was to create a single chip (one piece of silicon) with two or more processor cores. In other words, they replaced a single large processor with two half-sized processors, or four quarter-sized processors. This design allowed the same sized chip to produce less heat and offered the opportunity to permit multiple calculations to take place at the same time.

For the Processor Manager, multiple cores are more complex to manage than a single core. We'll discuss multiple core processing in Chapter 6.

Job Scheduling Versus Process Scheduling

The Processor Manager is a composite of two submanagers: one in charge of job scheduling and the other in charge of process scheduling. They're known as the **Job Scheduler** and the **Process Scheduler**.

Typically a user views a job either as a series of global job steps—compilation, loading, and execution—or as one all-encompassing step—execution. However, the scheduling of jobs is actually handled on two levels by most operating systems. If we return to the example presented earlier, we can see that a hierarchy exists between the Job Scheduler and the Process Scheduler.

The scheduling of the two jobs, to assemble the toy and to bandage the injury, was on a first-come, first-served and priority basis. Each job is initiated by the Job Scheduler based on certain criteria. Once a job is selected for execution, the Process Scheduler determines when each step, or set of steps, is executed—a decision that's also based on certain criteria. When you started assembling the toy, each step in the assembly instructions would have been selected for execution by the Process Scheduler.

Therefore, each job (or program) passes through a hierarchy of managers. Since the first one it encounters is the Job Scheduler, this is also called the **high-level scheduler**. It is only concerned with selecting jobs from a queue of incoming jobs and placing them in the process queue, whether batch or interactive, based on each job's characteristics. The Job Scheduler's goal is to put the jobs in a sequence that will use all of the system's resources as fully as possible.

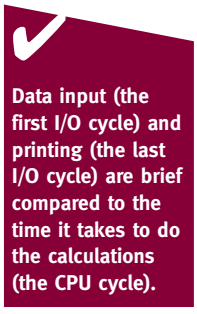
This is an important function. For example, if the Job Scheduler selected several jobs to run consecutively and each had a lot of I/O, then the I/O devices would be kept very busy. The CPU might be busy handling the I/O (if an I/O controller were not used) so little computation might get done. On the other hand, if the Job Scheduler selected several consecutive jobs with a great deal of computation, then the CPU would be very busy doing that. The I/O devices would be idle waiting for I/O requests. Therefore, the Job Scheduler strives for a balanced mix of jobs that require large amounts of I/O interaction and jobs that require large amounts of computation. Its goal is to keep most components of the computer system busy most of the time.

Process Scheduler

Most of this chapter is dedicated to the Process Scheduler because after a job has been placed on the READY queue by the Job Scheduler, the Process Scheduler takes over. It determines which jobs will get the CPU, when, and for how long. It also decides when processing should be interrupted, determines which queues the job should be moved to during its execution, and recognizes when a job has concluded and should be terminated.

The Process Scheduler is the **low-level scheduler** that assigns the CPU to execute the processes of those jobs placed on the READY queue by the Job Scheduler. This becomes a crucial function when the processing of several jobs has to be orchestrated—just as when you had to set aside your assembly and rush to help your neighbor.

To schedule the CPU, the Process Scheduler takes advantage of a common trait among most computer programs: they alternate between CPU cycles and I/O cycles. Notice that the following job has one relatively long CPU cycle and two very brief I/O cycles:



Data input (the first I/O cycle) and printing (the last I/O cycle) are brief compared to the time it takes to do the calculations (the CPU cycle).

```
{
printf("\nEnter the first integer: ");
scanf("%d", &a);
printf("\nEnter the second integer: ");
scanf("%d", &b);
```

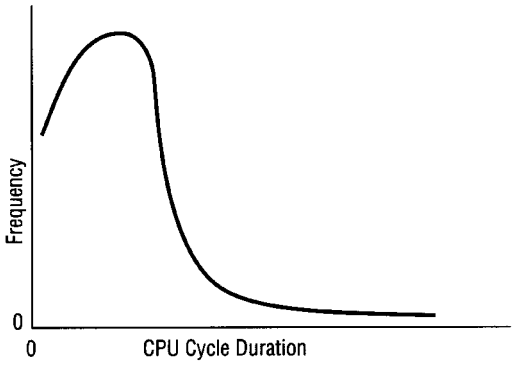
I/O cycle

```
c = a+b
d = (a*b)-c
e = a-b
f = d/e
```

CPU cycle

```
printf("\n a+b= %d", c);
printf("\n (a*b)-c = %d", d);
printf("\n a-b = %d", e);
printf("\n d/e = %d", f);
}
```

I/O cycle



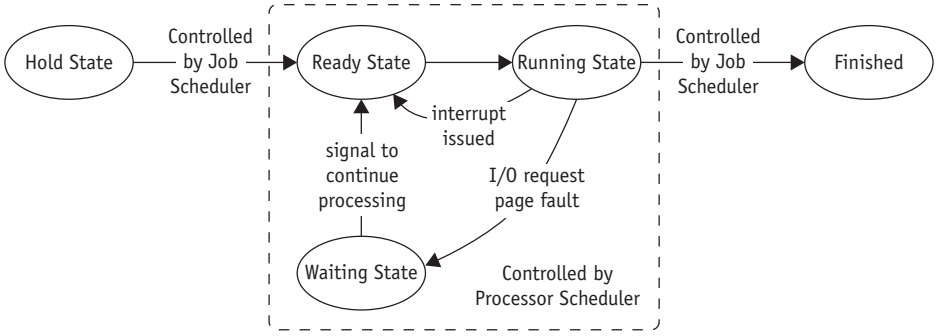
(figure 4.1)

Distribution of CPU cycle times. This distribution shows a greater number of jobs requesting short CPU cycles (the frequency peaks close to the low end of the CPU cycle axis), and fewer jobs requesting long CPU cycles.

Although the duration and frequency of CPU cycles vary from program to program, there are some general tendencies that can be exploited when selecting a scheduling algorithm. For example, **I/O-bound** jobs (such as printing a series of documents) have many brief CPU cycles and long I/O cycles, whereas **CPU-bound** jobs (such as finding the first 300 prime numbers) have long CPU cycles and shorter I/O cycles. The total effect of all CPU cycles, from both I/O-bound and CPU-bound jobs, approximates a Poisson distribution curve as shown in Figure 4.1.

In a highly interactive environment, there's also a third layer of the Processor Manager called the **middle-level scheduler**. In some cases, especially when the system is overloaded, the middle-level scheduler finds it is advantageous to remove active jobs from memory to reduce the degree of multiprogramming, which allows jobs to be completed faster. The jobs that are swapped out and eventually swapped back in are managed by the middle-level scheduler.

In a single-user environment, there's no distinction made between job and process scheduling because only one job is active in the system at any given time. So the CPU and all other resources are dedicated to that job, and to each of its processes in turn, until the job is completed.



(figure 4.2)

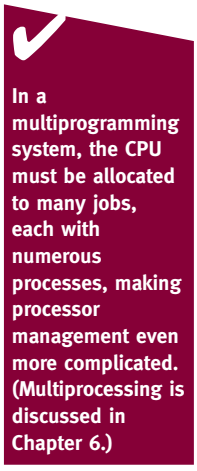
A typical job (or process) changes status as it moves through the system from HOLD to FINISHED.

Job and Process Status

As a job moves through the system, it's always in one of five states (or at least three) as it changes from HOLD to READY to RUNNING to WAITING and eventually to FINISHED as shown in Figure 4.2. These are called the **job status** or the **process status**.

Here's how the job status changes when a user submits a job to the system via batch or interactive mode. When the job is accepted by the system, it's put on HOLD and placed in a queue. In some systems, the job spooler (or disk controller) creates a table with the characteristics of each job in the queue and notes the important features of the job, such as an estimate of CPU time, priority, special I/O devices required, and maximum memory required. This table is used by the Job Scheduler to decide which job is to be run next.

From HOLD, the job moves to READY when it's ready to run but is waiting for the CPU. In some systems, the job (or process) might be placed on the READY list directly. RUNNING, of course, means that the job is being processed. In a single processor system, this is one "job" or process. WAITING means that the job can't continue until a specific resource is allocated or an I/O operation has finished. Upon completion, the job is FINISHED and returned to the user.



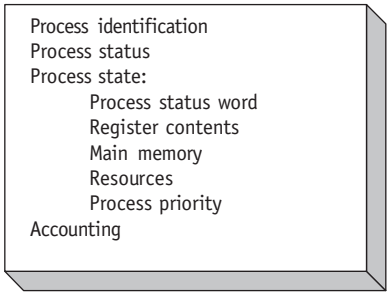
In a multiprogramming system, the CPU must be allocated to many jobs, each with numerous processes, making processor management even more complicated. (Multiprocessing is discussed in Chapter 6.)

The transition from one job or process status to another is initiated by either the Job Scheduler or the Process Scheduler:

- The transition from HOLD to READY is initiated by the Job Scheduler according to some predefined policy. At this point, the availability of enough main memory and any requested devices is checked.
- The transition from READY to RUNNING is handled by the Process Scheduler according to some predefined algorithm (i.e., FCFS, SJN, priority scheduling, SRT, or round robin—all of which will be discussed shortly).
- The transition from RUNNING back to READY is handled by the Process Scheduler according to some predefined time limit or other criterion, for example a priority interrupt.
- The transition from RUNNING to WAITING is handled by the Process Scheduler and is initiated by an instruction in the job such as a command to READ, WRITE, or other I/O request, or one that requires a page fetch.
- The transition from WAITING to READY is handled by the Process Scheduler and is initiated by a signal from the I/O device manager that the I/O request has been satisfied and the job can continue. In the case of a page fetch, the page fault handler will signal that the page is now in memory and the process can be placed on the READY queue.
- Eventually, the transition from RUNNING to FINISHED is initiated by the Process Scheduler or the Job Scheduler either when (1) the job is successfully completed and it ends execution or (2) the operating system indicates that an error has occurred and the job is being terminated prematurely.

Process Control Blocks

Each process in the system is represented by a data structure called a **Process Control Block** (**PCB**) that performs the same function as a traveler's passport. The PCB (illustrated in Figure 4.3) contains the basic information about the job, including what it is, where it's going, how much of its processing has been completed, where it's stored, and how much it has spent in using resources.



(figure 4.3)

Contents of each job's Process Control Block.

Process Identification

Each job is uniquely identified by the user's identification and a pointer connecting it to its descriptor (supplied by the Job Scheduler when the job first enters the system and is placed on HOLD).

Process Status

This indicates the current status of the job—HOLD, READY, RUNNING, or WAITING—and the resources responsible for that status.

Process State

This contains all of the information needed to indicate the current state of the job such as:

- *Process Status Word*—the current instruction counter and register contents when the job isn't running but is either on HOLD or is READY or WAITING. If the job is RUNNING, this information is left undefined.
- *Register Contents*—the contents of the register if the job has been interrupted and is waiting to resume processing.
- *Main Memory*—pertinent information, including the address where the job is stored and, in the case of virtual memory, the mapping between virtual and physical memory locations.

- *Resources*—information about all resources allocated to this job. Each resource has an identification field listing its type and a field describing details of its allocation, such as the sector address on a disk. These resources can be hardware units (disk drives or printers, for example) or files.
- *Process Priority*—used by systems using a priority scheduling algorithm to select which job will be run next.

Accounting

This contains information used mainly for billing purposes and performance measurement. It indicates what kind of resources the job used and for how long. Typical charges include:

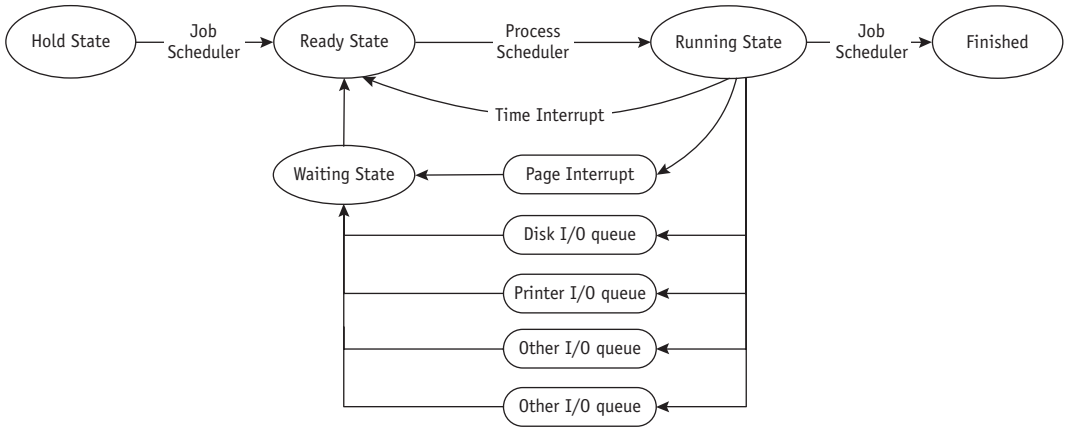
- Amount of CPU time used from beginning to end of its execution.
- Total time the job was in the system until it exited.
- Main storage occupancy—how long the job stayed in memory until it finished execution. This is usually a combination of time and space used; for example, in a paging system it may be recorded in units of page-seconds.
- Secondary storage used during execution. This, too, is recorded as a combination of time and space used.
- System programs used, such as compilers, editors, or utilities.
- Number and type of I/O operations, including I/O transmission time, that includes utilization of channels, control units, and devices.
- Time spent waiting for I/O completion.
- Number of input records read (specifically, those entered online or coming from optical scanners, card readers, or other input devices), and number of output records written.

PCBs and Queueing

A job's PCB is created when the Job Scheduler accepts the job and is updated as the job progresses from the beginning to the end of its execution.

Queues use PCBs to track jobs the same way customs officials use passports to track international visitors. The PCB contains all of the data about the job needed by the operating system to manage the processing of the job. As the job moves through the system, its progress is noted in the PCB.

The PCBs, not the jobs, are linked to form the queues as shown in Figure 4.4. Although each PCB is not drawn in detail, the reader should imagine each queue as a linked list of PCBs. The PCBs for every ready job are linked on the READY queue, and



(figure 4.4)

Queuing paths from HOLD to FINISHED. The Job and Processor schedulers release the resources when the job leaves the RUNNING state.

all of the PCBs for the jobs just entering the system are linked on the HOLD queue. The jobs that are WAITING, however, are linked together by "reason for waiting," so the PCBs for the jobs in this category are linked into several queues. For example, the PCBs for jobs that are waiting for I/O on a specific disk drive are linked together, while those waiting for the printer are linked in a different queue. These queues need to be managed in an orderly fashion and that's determined by the process scheduling policies and algorithms.

Process Scheduling Policies

In a multiprogramming environment, there are usually more jobs to be executed than could possibly be run at one time. Before the operating system can schedule them, it needs to resolve three limitations of the system: (1) there are a finite number of resources (such as disk drives, printers, and tape drives); (2) some resources, once they're allocated, can't be shared with another job (e.g., printers); and (3) some resources require operator intervention—that is, they can't be reassigned automatically from job to job (such as tape drives).

What's a good **process scheduling policy**? Several criteria come to mind, but notice in the list below that some contradict each other:

- *Maximize throughput*. Run as many jobs as possible in a given amount of time. This could be accomplished easily by running only short jobs or by running jobs without interruptions.

- *Minimize response time*. Quickly turn around interactive requests. This could be done by running only interactive jobs and letting the batch jobs wait until the interactive load ceases.
- *Minimize turnaround time*. Move entire jobs in and out of the system quickly. This could be done by running all batch jobs first (because batch jobs can be grouped to run more efficiently than interactive jobs).
- *Minimize waiting time*. Move jobs out of the READY queue as quickly as possible. This could only be done by reducing the number of users allowed on the system so the CPU would be available immediately whenever a job entered the READY queue.
- *Maximize CPU efficiency*. Keep the CPU busy 100 percent of the time. This could be done by running only CPU-bound jobs (and not I/O-bound jobs).
- *Ensure fairness for all jobs*. Give everyone an equal amount of CPU and I/O time. This could be done by not giving special treatment to any job, regardless of its processing characteristics or priority.

As we can see from this list, if the system favors one type of user then it hurts another or doesn't efficiently use its resources. The final decision rests with the system designer, who must determine which criteria are most important for that specific system. For example, you might decide to "maximize CPU utilization while minimizing response time and balancing the use of all system components through a mix of I/O-bound and CPU-bound jobs." So you would select the scheduling policy that most closely satisfies your criteria.

Although the Job Scheduler selects jobs to ensure that the READY and I/O queues remain balanced, there are instances when a job claims the CPU for a very long time before issuing an I/O request. If I/O requests are being satisfied (this is done by an I/O controller and will be discussed later), this extensive use of the CPU will build up the READY queue while emptying out the I/O queues, which creates an unacceptable imbalance in the system.

To solve this problem, the Process Scheduler often uses a timing mechanism and periodically interrupts running processes when a predetermined slice of time has expired. When that happens, the scheduler suspends all activity on the job currently running and reschedules it into the READY queue; it will be continued later. The CPU is now allocated to another job that runs until one of three things happens: the timer goes off, the job issues an I/O command, or the job is finished. Then the job moves to the READY queue, the WAIT queue, or the FINISHED queue, respectively. An I/O request is called a **natural wait** in multiprogramming environments (it allows the processor to be allocated to another job).

A scheduling strategy that interrupts the processing of a job and transfers the CPU to another job is called a **preemptive scheduling policy**; it is widely used in time-sharing environments. The alternative, of course, is a **nonpreemptive scheduling policy**, which functions without external interrupts (interrupts external to the job). Therefore, once a job captures the processor and begins execution, it remains in the RUNNING state

uninterrupted until it issues an I/O request (natural wait) or until it is finished (with exceptions made for infinite loops, which are interrupted by both preemptive and nonpreemptive policies).

Process Scheduling Algorithms

The Process Scheduler relies on a **process scheduling algorithm**, based on a specific policy, to allocate the CPU and move jobs through the system. Early operating systems used nonpreemptive policies designed to move batch jobs through the system as efficiently as possible. Most current systems, with their emphasis on interactive use and **response time**, use an algorithm that takes care of the immediate requests of interactive users.

Here are six process scheduling algorithms that have been used extensively.

First-Come, First-Served

First-come, first-served (FCFS) is a nonpreemptive scheduling algorithm that handles jobs according to their arrival time: the earlier they arrive, the sooner they're served. It's a very simple algorithm to implement because it uses a FIFO queue. This algorithm is fine for most batch systems, but it is unacceptable for interactive systems because interactive users expect quick response times.

With FCFS, as a new job enters the system its PCB is linked to the end of the READY queue and it is removed from the front of the queue when the processor becomes available—that is, after it has processed all of the jobs before it in the queue.

In a strictly FCFS system there are no WAIT queues (each job is run to completion), although there may be systems in which control (context) is switched on a natural wait (I/O request) and then the job resumes on I/O completion.

The following examples presume a strictly FCFS environment (no multiprogramming). **Turnaround time** is unpredictable with the FCFS policy; consider the following three jobs:

- Job A has a CPU cycle of 15 milliseconds.
- Job B has a CPU cycle of 2 milliseconds.
- Job C has a CPU cycle of 1 millisecond.

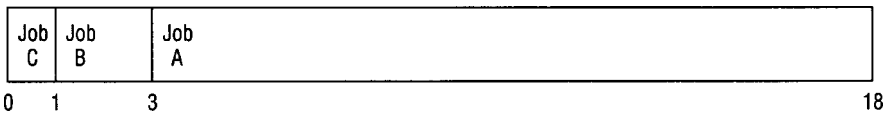
For each job, the CPU cycle contains both the actual CPU usage and the I/O requests. That is, it is the total run time. Using an FCFS algorithm with an arrival sequence of A, B, C, the timeline is shown in Figure 4.5.

(figure 4.5) *Timeline for job sequence A, B, C using the FCFS algorithm.*

If all three jobs arrive almost simultaneously, we can calculate that the turnaround time for Job A is 15, for Job B is 17, and for Job C is 18. So the average turnaround time is:

$$\frac{15 + 17 + 18}{3} = 16.67$$

However, if the jobs arrived in a different order, say C, B, A, then the results using the same FCFS algorithm would be as shown in Figure 4.6.

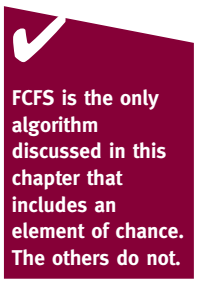


(figure 4.6) *Timeline for job sequence C, B, A using the FCFS algorithm.*

In this example the turnaround time for Job A is 18, for Job B is 3, and for Job C is 1 and the average turnaround time is:

$$\frac{18 + 3 + 1}{3} = 7.3$$

That's quite an improvement over the first sequence. Unfortunately, these two examples illustrate the primary disadvantage of using the FCFS concept—the average turnaround times vary widely and are seldom minimized. In fact, when there are three jobs in the READY queue, the system has only a 1 in 6 chance of running the jobs in the most advantageous sequence (C, B, A). With four jobs the odds fall to 1 in 24, and so on.



FCFS is the only algorithm discussed in this chapter that includes an element of chance. The others do not.

If one job monopolizes the system, the extent of its overall effect on system performance depends on the scheduling policy and whether the job is CPU-bound or I/O-bound. While a job with a long CPU cycle (in this example, Job A) is using the CPU, the other jobs in the system are waiting for processing or finishing their I/O requests (if an I/O controller is used) and joining the READY queue to wait for their turn to use the processor. If the I/O requests are not being serviced, the I/O queues would remain stable while the READY list grew (with new arrivals). In extreme cases, the READY queue could fill to capacity while the I/O queues would be empty, or stable, and the I/O devices would sit idle.

On the other hand, if the job is processing a lengthy I/O cycle, the I/O queues quickly build to overflowing and the CPU could be sitting idle (if an I/O controller is used). This situation is eventually resolved when the I/O-bound job finishes its I/O cycle, the queues start moving again, and the system can recover from the bottleneck.

In a strictly FCFS algorithm, neither situation occurs. However, the turnaround time is variable (unpredictable). For this reason, FCFS is a less attractive algorithm than one that would serve the shortest job first, as the next scheduling algorithm does, even in a nonmultiprogramming environment.

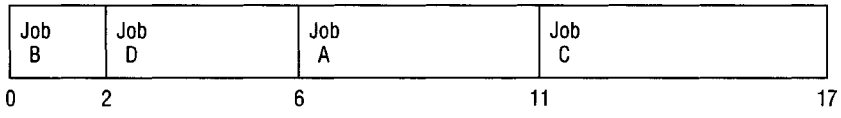
Shortest Job Next

Shortest job next (**SJN**) is a nonpreemptive scheduling algorithm (also known as shortest job first, or SJF) that handles jobs based on the length of their CPU cycle time. It's easiest to implement in batch environments where the estimated CPU time required to run the job is given in advance by each user at the start of each job. However, it doesn't work in interactive systems because users don't estimate in advance the CPU time required to run their jobs.

For example, here are four batch jobs, all in the READY queue, for which the CPU cycle, or run time, is estimated as follows:

Job:	A	B	C	D
CPU cycle:	5	2	6	4

The SJN algorithm would review the four jobs and schedule them for processing in this order: B, D, A, C. The timeline is shown in Figure 4.7.



(figure 4.7)

Timeline for job sequence B, D, A, C using the SJN algorithm.

The average turnaround time is:

$$\frac{2 + 6 + 11 + 17}{4} = 9.0$$

Let's take a minute to see why this algorithm can be proved to be optimal and will consistently give the minimum average turnaround time. We'll use the previous example to derive a general formula.

If we look at Figure 4.7, we can see that Job B finishes in its given time (2), Job D finishes in its given time plus the time it waited for B to run (4 + 2), Job A finishes in its given time plus D's time plus B's time (5 + 4 + 2), and Job C finishes in its given time plus that of the previous three (6 + 5 + 4 + 2). So when calculating the average we have:

$$\frac{(2) + (4 + 2) + (5 + 4 + 2) + (6 + 5 + 4 + 2)}{4} = 9.0$$

As you can see, the time for the first job appears in the equation four times—once for each job. Similarly, the time for the second job appears three times (the number of jobs minus one). The time for the third job appears twice (number of jobs minus 2) and the time for the fourth job appears only once (number of jobs minus 3).

So the above equation can be rewritten as:

$$\frac{4 * 2 + 3 * 4 + 2 * 5 + 1 * 6}{4} = 9.0$$

Because the time for the first job appears in the equation four times, it has four times the effect on the average time than does the length of the fourth job, which appears only once. Therefore, if the first job requires the shortest computation time, followed in turn by the other jobs, ordered from shortest to longest, then the result will be the smallest possible average. The formula for the average is as follows

$$\frac{t_1(n) + t_2(n - 1) + t_3(n - 2) + \ ... + t_n(n(1))}{n}$$

where n is the number of jobs in the queue and $t_j(j = 1, 2, 3,\ldots,n)$ is the length of the CPU cycle for each of the jobs.

However, the SJN algorithm is optimal only when all of the jobs are available at the same time and the CPU estimates are available and accurate.

Priority Scheduling

Priority scheduling is a nonpreemptive algorithm and one of the most common scheduling algorithms in batch systems, even though it may give slower turnaround to some users. This algorithm gives preferential treatment to important jobs. It allows the programs with the highest priority to be processed first, and they aren't interrupted until their CPU cycles (run times) are completed or a natural wait occurs. If two or more jobs with equal priority are present in the READY queue, the processor is allocated to the one that arrived first (first-come, first-served within priority).

Priorities can be assigned by a system administrator using characteristics extrinsic to the jobs. For example, they can be assigned based on the position of the user (researchers first, students last) or, in commercial environments, they can be purchased by the users who pay more for higher priority to guarantee the fastest possible processing of their jobs. With a priority algorithm, jobs are usually linked to one of several READY queues by the Job Scheduler based on their priority so the Process Scheduler manages multiple READY queues instead of just one. Details about multiple queues are presented later in this chapter.

Priorities can also be determined by the Processor Manager based on characteristics intrinsic to the jobs such as:

- *Memory requirements.* Jobs requiring large amounts of memory could be allocated lower priorities than those requesting small amounts of memory, or vice versa.
- *Number and type of peripheral devices.* Jobs requiring many peripheral devices would be allocated lower priorities than those requesting fewer devices.
- *Total CPU time.* Jobs having a long CPU cycle, or estimated run time, would be given lower priorities than those having a brief estimated run time.
- *Amount of time already spent in the system.* This is the total amount of elapsed time since the job was accepted for processing. Some systems increase the priority of jobs that have been in the system for an unusually long time to expedite their exit. This is known as **aging**.

These criteria are used to determine default priorities in many systems. The default priorities can be overruled by specific priorities named by users.

There are also preemptive priority schemes. These will be discussed later in this chapter in the section on multiple queues.

Shortest Remaining Time

Shortest remaining time (SRT) is the preemptive version of the SJN algorithm. The processor is allocated to the job closest to completion—but even this job can be preempted if a newer job in the READY queue has a time to completion that's shorter.

This algorithm can't be implemented in an interactive system because it requires advance knowledge of the CPU time required to finish each job. It is often used in batch environments, when it is desirable to give preference to short jobs, even though SRT involves more overhead than SJN because the operating system has to frequently monitor the CPU time for all the jobs in the READY queue and must perform context switching for the jobs being swapped (switched) at preemption time (not necessarily swapped out to the disk, although this might occur as well).

The example in Figure 4.8 shows how the SRT algorithm works with four jobs that arrived in quick succession (one CPU cycle apart).

If several jobs have the same amount of time remaining, the job that has been waiting the longest goes next. In other words, it uses the FCFS algorithm to break the tie.

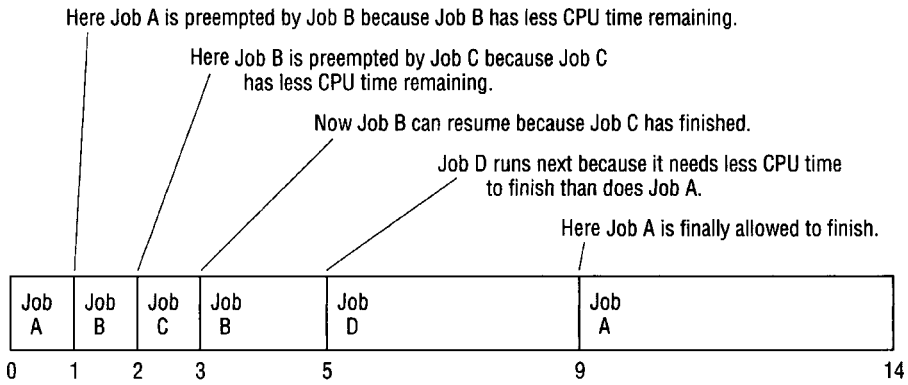
Arrival time:	0	1	2	3
Job:	A	B	C	D
CPU cycle:	6	3	1	4

In this case, the turnaround time is the completion time of each job minus its arrival time:

Job:	A	B	C	D
Turnaround:	14	4	1	6

So the average turnaround time is:

$$\frac{14 + 4 + 1 + 6}{4} = 6.25$$



(figure 4.8) *Timeline for job sequence A, B, C, D using the preemptive SRT algorithm. Each job is interrupted after one CPU cycle if another job is waiting with less CPU time remaining.*

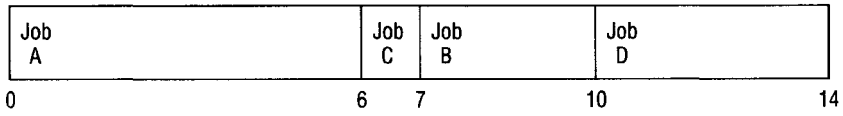
How does that compare to the same problem using the nonpreemptive SJN policy? Figure 4.9 shows the same situation using SJN.

In this case, the turnaround time is:

Job:	A	B	C	D
Turnaround:	6	9	5	11

So the average turnaround time is:

$$\frac{6 + 9 + 5 + 11}{4} = 7.75$$



(figure 4.9) *Timeline for the same job sequence A, B, C, D using the nonpreemptive SJN algorithm.*

Note in Figure 4.9 that initially A is the only job in the READY queue so it runs first and continues until it's finished because SJN is a nonpreemptive algorithm. The next job to be run is C because when Job A is finished (at time 6), all of the other jobs (B, C, and D) have arrived. Of those three, C has the shortest CPU cycle, so it is the next one run, then B, and finally D.

Therefore, with this example, SRT at 6.25 is faster than SJN at 7.75. However, we neglected to include the time required by the SRT algorithm to do the context switching. **Context switching** is required by all preemptive algorithms. When Job A is preempted, all of its processing information must be saved in its PCB for later, when Job A's execution is to be continued, and the contents of Job B's PCB are loaded into the appropriate registers so it can start running again; this is a context switch. Later, when Job A is once again assigned to the processor, another context switch is performed. This time the information from the preempted job is stored in its PCB, and the contents of Job A's PCB are loaded into the appropriate registers.

How the context switching is actually done depends on the architecture of the CPU; in many systems, there are special instructions that provide quick saving and restoring of information. The switching is designed to be performed efficiently but, no matter how fast it is, it still takes valuable CPU time. So although SRT appears to be faster, in a real operating environment its advantages are diminished by the time spent in context switching. A precise comparison of SRT and SJN would have to include the time required to do the context switching.

Round Robin

Round robin is a preemptive process scheduling algorithm that is used extensively in interactive systems. It's easy to implement and isn't based on job characteristics but on a predetermined slice of time that's given to each job to ensure that the CPU is equally shared among all active processes and isn't monopolized by any one job.

This time slice is called a **time quantum** and its size is crucial to the performance of the system. It usually varies from 100 milliseconds to 1 or 2 seconds.

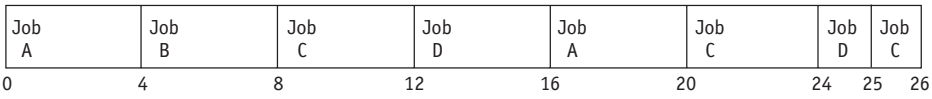
Jobs are placed in the READY queue using a first-come, first-served scheme and the Process Scheduler selects the first job from the front of the queue, sets the timer to the time quantum, and allocates the CPU to this job. If processing isn't finished when time expires, the job is preempted and put at the end of the READY queue and its information is saved in its PCB.

In the event that the job's CPU cycle is shorter than the time quantum, one of two actions will take place: (1) If this is the job's last CPU cycle and the job is finished, then all resources allocated to it are released and the completed job is returned to the user;

(2) if the CPU cycle has been interrupted by an I/O request, then information about the job is saved in its PCB and it is linked at the end of the appropriate I/O queue. Later, when the I/O request has been satisfied, it is returned to the end of the READY queue to await allocation of the CPU.

The example in Figure 4.10 illustrates a round robin algorithm with a time slice of 4 milliseconds (I/O requests are ignored):

Arrival time:	0	1	2	3
Job:	A	B	C	D
CPU cycle:	8	4	9	5



(figure 4.10) *Timeline for job sequence A, B, C, D using the preemptive round robin algorithm with time slices of 4 ms.*

The turnaround time is the completion time minus the arrival time:

Job:	A	B	C	D
Turnaround:	20	7	24	22

So the average turnaround time is:

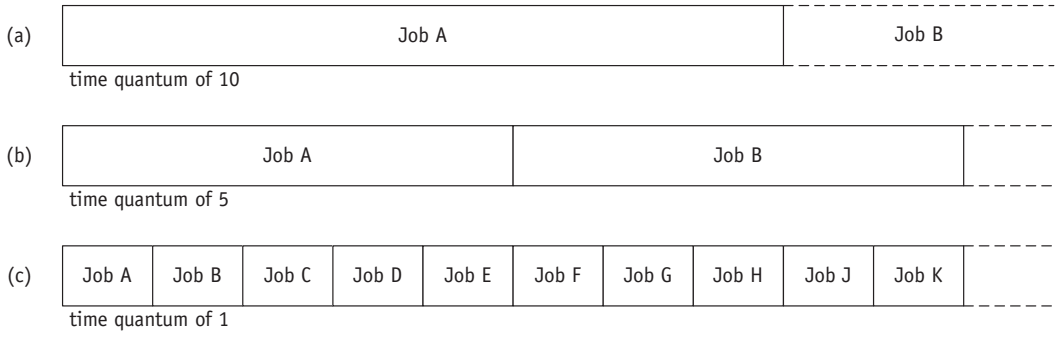
$$\frac{20 + 7 + 24 + 22}{4} = 18.25$$

Note that in Figure 4.10, Job A was preempted once because it needed 8 milliseconds to complete its CPU cycle, while Job B terminated in one time quantum. Job C was preempted twice because it needed 9 milliseconds to complete its CPU cycle, and Job D was preempted once because it needed 5 milliseconds. In their last execution or swap into memory, both Jobs D and C used the CPU for only 1 millisecond and terminated before their last time quantum expired, releasing the CPU sooner.

With round robin and a queue with numerous processes, each process will get access to the processor before the first process will get access a second time.

The efficiency of round robin depends on the size of the time quantum in relation to the average CPU cycle. If the quantum is too large—that is, if it's larger than most CPU cycles—then the algorithm reduces to the FCFS scheme. If the quantum is too small, then the amount of context switching slows down the execution of the jobs and the amount of overhead is dramatically increased, as the three examples in Figure 4.11 demonstrate. Job A has a CPU cycle of 8 milliseconds. The amount of context switching increases as the time quantum decreases in size.

In Figure 4.11, the first case (a) has a time quantum of 10 milliseconds and there is no context switching (and no overhead). The CPU cycle ends shortly before the time



(figure 4.11)

Context switches for three different time quantums. In (a), Job A (which requires only 8 cycles to run to completion) finishes before the time quantum of 10 expires. In (b) and (c), the time quantum expires first, interrupting the jobs.

quantum expires and the job runs to completion. For this job with this time quantum, there is no difference between the round robin algorithm and the FCFS algorithm.

In the second case (b), with a time quantum of 5 milliseconds, there is one context switch. The job is preempted once when the time quantum expires, so there is some overhead for context switching and there would be a delayed turnaround based on the number of other jobs in the system.

In the third case (c), with a time quantum of 1 millisecond, there are 10 context switches because the job is preempted every time the time quantum expires; overhead becomes costly and turnaround time suffers accordingly.

What's the best time quantum size? The answer should be predictable by now: it depends on the system. If it's an interactive environment, the system is expected to respond quickly to its users, especially when they make simple requests. If it's a batch system, response time is not a factor (turnaround is) and overhead becomes very important.

Here are two general rules of thumb for selecting the proper time quantum: (1) it should be long enough to allow 80 percent of the CPU cycles to run to completion, and (2) it should be at least 100 times longer than the time required to perform one context switch. These rules are used in some systems, but they are not inflexible.

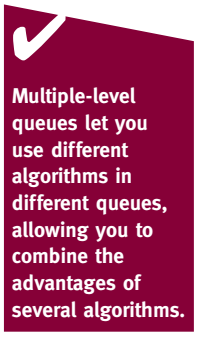
Multiple-Level Queues

Multiple-level queues isn't really a separate scheduling algorithm but works in conjunction with several of the schemes already discussed and is found in systems with jobs that can be grouped according to a common characteristic. We've already introduced at least one kind of multiple-level queue—that of a priority-based system with different queues for each priority level.

Another kind of system might gather all of the CPU-bound jobs in one queue and all I/O-bound jobs in another. The Process Scheduler then alternately selects jobs from each queue to keep the system balanced.

A third common example is one used in a hybrid environment that supports both batch and interactive jobs. The batch jobs are put in one queue called the background queue while the interactive jobs are put in a foreground queue and are treated more favorably than those on the background queue.

All of these examples have one thing in common: The scheduling policy is based on some predetermined scheme that allocates special treatment to the jobs in each queue. Within each queue, the jobs are served in FCFS fashion.



Multiple-level queues let you use different algorithms in different queues, allowing you to combine the advantages of several algorithms.

Multiple-level queues raise some interesting questions:

- Is the processor allocated to the jobs in the first queue until it is empty before moving to the next queue, or does it travel from queue to queue until the last job on the last queue has been served and then go back to serve the first job on the first queue, or something in between?
- Is this fair to those who have earned, or paid for, a higher priority?
- Is it fair to those in a low-priority queue?
- If the processor is allocated to the jobs on the first queue and it never empties out, when will the jobs in the last queues be served?
- Can the jobs in the last queues get "time off for good behavior" and eventually move to better queues?

The answers depend on the policy used by the system to service the queues. There are four primary methods to the movement: not allowing movement between queues, moving jobs from queue to queue, moving jobs from queue to queue and increasing the time quantums for lower queues, and giving special treatment to jobs that have been in the system for a long time (aging).

Case 1: No Movement Between Queues

No movement between queues is a very simple policy that rewards those who have high-priority jobs. The processor is allocated to the jobs in the high-priority queue in FCFS fashion and it is allocated to jobs in low-priority queues only when the high-priority queues are empty. This policy can be justified if there are relatively few users with high-priority jobs so the top queues quickly empty out, allowing the processor to spend a fair amount of time running the low-priority jobs.

Case 2: Movement Between Queues

Movement between queues is a policy that adjusts the priorities assigned to each job: High-priority jobs are treated like all the others once they are in the system. (Their initial priority may be favorable.) When a time quantum interrupt occurs, the job is preempted and moved to the end of the next lower queue. A job may also have its priority increased; for example, when it issues an I/O request before its time quantum has expired.

This policy is fairest in a system in which the jobs are handled according to their computing cycle characteristics: CPU-bound or I/O-bound. This assumes that a job that exceeds its time quantum is CPU-bound and will require more CPU allocation than one that requests I/O before the time quantum expires. Therefore, the CPU-bound jobs are placed at the end of the next lower-level queue when they're preempted because of the expiration of the time quantum, while I/O-bound jobs are returned to the end of the next higher-level queue once their I/O request has finished. This facilitates I/O-bound jobs and is good in interactive systems.

Case 3: Variable Time Quantum Per Queue

Variable time quantum per queue is a variation of the movement between queues policy, and it allows for faster turnaround of CPU-bound jobs.

In this scheme, each of the queues is given a time quantum twice as long as the previous queue. The highest queue might have a time quantum of 100 milliseconds. So the second-highest queue would have a time quantum of 200 milliseconds, the third would have 400 milliseconds, and so on. If there are enough queues, the lowest one might have a relatively long time quantum of 3 seconds or more.

If a job doesn't finish its CPU cycle in the first time quantum, it is moved to the end of the next lower-level queue; and when the processor is next allocated to it, the job executes

for twice as long as before. With this scheme a CPU-bound job can execute for longer and longer periods of time, thus improving its chances of finishing faster.

Case 4: Aging

Aging is used to ensure that jobs in the lower-level queues will eventually complete their execution. The operating system keeps track of each job's waiting time and when a job gets too old—that is, when it reaches a certain time limit—the system moves the job to the next highest queue, and so on until it reaches the top queue. A more drastic aging policy is one that moves the old job directly from the lowest queue to the end of the top queue. Regardless of its actual implementation, an aging policy guards against the indefinite postponement of unwieldy jobs. As you might expect, **indefinite postponement** means that a job's execution is delayed for an undefined amount of time because it is repeatedly preempted so other jobs can be processed. (We all know examples of an unpleasant task that's been indefinitely postponed to make time for a more appealing pastime). Eventually the situation could lead to the old job's starvation. Indefinite postponement is a major problem when allocating resources and one that will be discussed in detail in Chapter 5.

A Word About Interrupts

We first encountered **interrupts** in Chapter 3 when the Memory Manager issued page interrupts to accommodate job requests. In this chapter we examined another type of interrupt that occurs when the time quantum expires and the processor is deallocated from the running job and allocated to another one.

There are other interrupts that are caused by events internal to the process. I/O interrupts are issued when a READ or WRITE command is issued. (We'll explain them in detail in Chapter 7.) Internal interrupts, or synchronous interrupts, also occur as a direct result of the arithmetic operation or job instruction currently being processed.

Illegal arithmetic operations, such as the following, can generate interrupts:

- Attempts to divide by zero
- Floating-point operations generating an overflow or underflow
- Fixed-point addition or subtraction that causes an arithmetic overflow

Illegal job instructions, such as the following, can also generate interrupts:

- Attempts to access protected or nonexistent storage locations
- Attempts to use an undefined operation code
- Operating on invalid data
- Attempts to make system changes, such as trying to change the size of the time quantum

The control program that handles the interruption sequence of events is called the **interrupt handler**. When the operating system detects a nonrecoverable error, the interrupt handler typically follows this sequence:

1. The type of interrupt is described and stored—to be passed on to the user as an error message.
2. The state of the interrupted process is saved, including the value of the program counter, the mode specification, and the contents of all registers.
3. The interrupt is processed: The error message and state of the interrupted process are sent to the user; program execution is halted; any resources allocated to the job are released; and the job exits the system.
4. The processor resumes normal operation.

If we're dealing with internal interrupts only, which are nonrecoverable, the job is terminated in Step 3. However, when the interrupt handler is working with an I/O interrupt, time quantum, or other recoverable interrupt, Step 3 simply halts the job and moves it to the appropriate I/O device queue, or READY queue (on time out). Later, when the I/O request is finished, the job is returned to the READY queue. If it was a time out (quantum interrupt), the job (or process) is already on the READY queue.

Conclusion

The Processor Manager must allocate the CPU among all the system's users. In this chapter we've made the distinction between job scheduling, the selection of incoming jobs based on their characteristics, and process scheduling, the instant-by-instant allocation of the CPU. We've also described how interrupts are generated and resolved by the interrupt handler.

Each scheduling algorithm presented in this chapter has unique characteristics, objectives, and applications. A system designer can choose the best policy and algorithm only after carefully evaluating their strengths and weaknesses. Table 4.1 shows how the algorithms presented in this chapter compare.

In the next chapter we'll explore the demands placed on the Processor Manager as it attempts to synchronize execution of all the jobs in the system.

Algorithm	Policy Type	Best for	Disadvantages	Advantages
FCFS	Nonpreemptive	Batch	Unpredictable turnaround times	Easy to implement
SJN	Nonpreemptive	Batch	Indefinite postponement of some jobs	Minimizes average waiting time
Priority scheduling	Nonpreemptive	Batch	Indefinite postponement of some jobs	Ensures fast completion of important jobs
SRT	Preemptive	Batch	Overhead incurred by context switching	Ensures fast completion of short jobs
Round robin	Preemptive	Interactive	Requires selection of good time quantum	Provides reasonable response times to interactive users; provides fair CPU allocation
Multiple-level queues	Preemptive/ Nonpreemptive	Batch/ interactive	Overhead incurred by monitoring of queues	Flexible scheme; counteracts indefinite postponement with aging or other queue movement; gives fair treatment to CPU-bound jobs by incrementing time quantums on lower-priority queues or other queue movement

(table 4.1)

Comparison of the scheduling algorithms discussed in this chapter.

Key Terms

aging: a policy used to ensure that jobs that have been in the system for a long time in the lower-level queues will eventually complete their execution.

context switching: the acts of saving a job's processing information in its PCB so the job can be swapped out of memory and of loading the processing information from the PCB of another job into the appropriate registers so the CPU can process it. Context switching occurs in all preemptive policies.

CPU-bound: a job that will perform a great deal of nonstop processing before issuing an interrupt.

first-come, first-served (FCFS): a nonpreemptive process scheduling policy (or algorithm) that handles jobs according to their arrival time.

high-level scheduler: a synonym for the Job Scheduler.

I/O-bound: a job that requires a large number of input/output operations, resulting in too much free time for the CPU.

indefinite postponement: signifies that a job's execution is delayed indefinitely because it is repeatedly preempted so other jobs can be processed.

interrupt: a hardware signal that suspends execution of a program and activates the execution of a special program known as the interrupt handler.

interrupt handler: the program that controls what action should be taken by the operating system when a sequence of events is interrupted.

Job Scheduler: the high-level scheduler of the Processor Manager that selects jobs from a queue of incoming jobs based on each job's characteristics.

job status: the condition of a job as it moves through the system from the beginning to the end of its execution.

low-level scheduler: a synonym for the Process Scheduler.

middle-level scheduler: a scheduler used by the Processor Manager when the system to remove active processes from memory becomes overloaded. The middle-level scheduler swaps these processes back into memory when the system overload has cleared.

multiple-level queues: a process scheduling scheme (used with other scheduling algorithms) that groups jobs according to a common characteristic.

multiprogramming: a technique that allows a single processor to process several programs residing simultaneously in main memory and interleaving their execution by overlapping I/O requests with CPU requests.

natural wait: a common term used to identify an I/O request from a program in a multiprogramming environment that would cause a process to wait "naturally" before resuming execution.

nonpreemptive scheduling policy: a job scheduling strategy that functions without external interrupts so that once a job captures the processor and begins execution, it remains in the running state uninterrupted until it issues an I/O request or it's finished.

preemptive scheduling policy: any process scheduling strategy that, based on predetermined policies, interrupts the processing of a job and transfers the CPU to another job. It is widely used in time-sharing environments.

priority scheduling: a nonpreemptive process scheduling policy (or algorithm) that allows for the execution of high-priority jobs before low-priority jobs.

process: an instance of execution of a program that is identifiable and controllable by the operating system.

Process Control Block (PCB): a data structure that contains information about the current status and characteristics of a process.

Process Scheduler: the low-level scheduler of the Processor Manager that establishes the order in which processes in the READY queue will be served by the CPU.

process scheduling algorithm: an algorithm used by the Job Scheduler to allocate the CPU and move jobs through the system.

process scheduling policy: any policy used by the Processor Manager to select the order in which incoming jobs will be executed.

process status: information stored in the job's PCB that indicates the current position of the job and the resources responsible for that status.

processor: (1) a synonym for the CPU, or (2) any component in a computing system capable of performing a sequence of activities.

program: an interactive unit, such as a file stored on a disk.

queue: a linked list of PCBs that indicates the order in which jobs or processes will be serviced.

response time: a measure of the efficiency of an interactive system that tracks the speed with which the system will respond to a user's command.

round robin: a preemptive process scheduling policy (or algorithm) that allocates to each job one unit of processing time per turn to ensure that the CPU is equally shared among all active processes and isn't monopolized by any one job.

shortest job next (SJN): a nonpreemptive process scheduling policy (or algorithm) that selects the waiting job with the shortest CPU cycle time.

shortest remaining time (SRT): a preemptive process scheduling policy (or algorithm) similar to the SJN algorithm that allocates the processor to the job closest to completion.

task: (1) the term used to describe a process, or (2) the basic unit of concurrent programming languages that defines a sequence of instructions that may be executed in parallel with other similar units.

thread: a portion of a program that can run independently of other portions. Multithreaded applications programs can have several threads running at one time with the same or different priorities.

time quantum: a period of time assigned to a process for execution before it is preempted.

turnaround time: a measure of a system's efficiency that tracks the time required to execute a job and return output to the user.

Interesting Searches

- CPU Cycle Time
- Task Control Block (TCB)
- Processor Bottleneck
- Processor Queue Length
- I/O Interrupts

Exercises

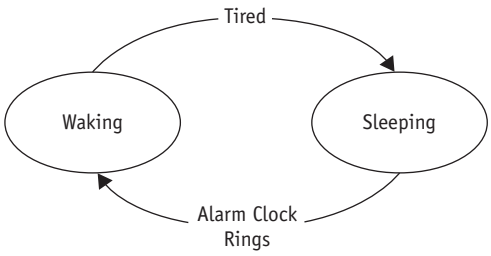
Research Topics

A. Multi-core technology can often, but not necessarily always, make applications run faster. Research some real-life computing environments that are expected to benefit from multi-core chips and briefly explain why. Cite your academic sources.

B. Compare two processors currently being produced for personal computers. Use standard industry benchmarks for your comparison and briefly list the advantages and disadvantages of each. You can compare different processors from the same manufacturer (such as two Intel processors) or different processors from different manufacturers (such as Intel and AMD).

Exercises

1. Figure 4.12 is a simplified process model of you, in which there are only two states: sleeping and waking. You make the transition from waking to sleeping when you are tired, and from sleeping to waking when the alarm clock goes off.
   a. Add three more states to the diagram (for example, one might be eating).
   b. State all of the possible transitions among the five states.



(figure 4.12)

Process model of two states.

2. Describe context switching in lay terms and identify the process information that needs to be saved, changed, or updated when context switching takes place.
3. Five jobs (A, B, C, D, E) are already in the READY queue waiting to be processed. Their estimated CPU cycles are respectively: 2, 10, 15, 6, and 8. Using SJN, in what order should they be processed?
4. A job running in a system, with variable time quantums per queue, needs 30 milliseconds to run to completion. If the first queue has a time quantum of 5 milliseconds and each queue thereafter has a time quantum that is twice as large as the previous one, how many times will the job be interrupted and on which queue will it finish its execution?
5. Describe the advantages of having a separate queue for Print I/O and for Disk I/O as illustrated in Figure 4.4.
6. Given the following information:

Job	Arrival Time	CPU Cycle
A	0	2
B	1	12
C	2	4
D	4	1
E	5	8
F	7	5
G	8	3

Using SJN, draw a timeline showing the time that each job arrives and the order that each is processed. Calculate the finish time for each job.

7. Given the following information:

Job	Arrival Time	CPU Cycle
A	0	10
B	2	12
C	3	3
D	6	1
E	9	15

Draw a timeline for each of the following scheduling algorithms. (It may be helpful to first compute a start and finish time for each job.)

a. FCFS

b. SJN

c. SRT

d. Round robin (using a time quantum of 5, ignore context switching and natural wait)

8. Using the same information from Exercise 7, calculate which jobs will have arrived ready for processing by the time the first job is finished or interrupted using each of the following scheduling algorithms.

a. FCFS

b. SJN

c. SRT

d. Round robin (using a time quantum of 5, ignore context switching and natural wait)

9. Using the same information given for Exercise 7, compute the waiting time and turnaround time for every job for each of the following scheduling algorithms (ignoring context switching overhead).

a. FCFS

b. SJN

c. SRT

d. Round robin (using a time quantum of 2)

Advanced Exercises

10. Consider a variation of round robin in which a process that has used its full time quantum is returned to the end of the READY queue, while one that has used half of its time quantum is returned to the middle of the queue and one

that has used one-fourth of its time quantum goes to a place one-fourth of the distance away from the beginning of the queue.

a. What is the objective of this scheduling policy?

b. Discuss the advantage and disadvantage of its implementation.

11. In a single-user dedicated system, such as a personal computer, it's easy for the user to determine when a job is caught in an infinite loop. The typical solution to this problem is for the user to manually intervene and terminate the job. What mechanism would you implement in the Process Scheduler to automate the termination of a job that's in an infinite loop? Take into account jobs that legitimately use large amounts of CPU time; for example, one "finding the first 10,000 prime numbers."
12. Some guidelines for selecting the right time quantum were given in this chapter. As a system designer, how would you know when you have chosen the best time quantum? What factors would make this time quantum best from the user's point of view? What factors would make this time quantum best from the system's point of view?
13. Using the process state diagrams of Figure 4.2, explain why there's no transition:

a. From the READY state to the WAITING state

b. From the WAITING state to the RUNNING state

Programming Exercises

14. Write a program that will simulate FCFS, SJN, SRT, and round robin scheduling algorithms. For each algorithm, the program should compute waiting time and turnaround time of every job as well as the average waiting time and average turnaround time. The average values should be consolidated in a table for easy comparison. You may use the following data to test your program. The time quantum for round robin is 4 milliseconds and the context switching time is 0.

Arrival Time	CPU Cycle (in milliseconds)
0	6
3	2
5	1
9	7
10	5
12	3

14	4
16	5
17	7
19	2

15. Using your program from Exercise 14, change the context switching time to 0.4 milliseconds. Compare outputs from both runs and discuss which would be the better policy. Describe any drastic changes encountered or a lack of changes and why.