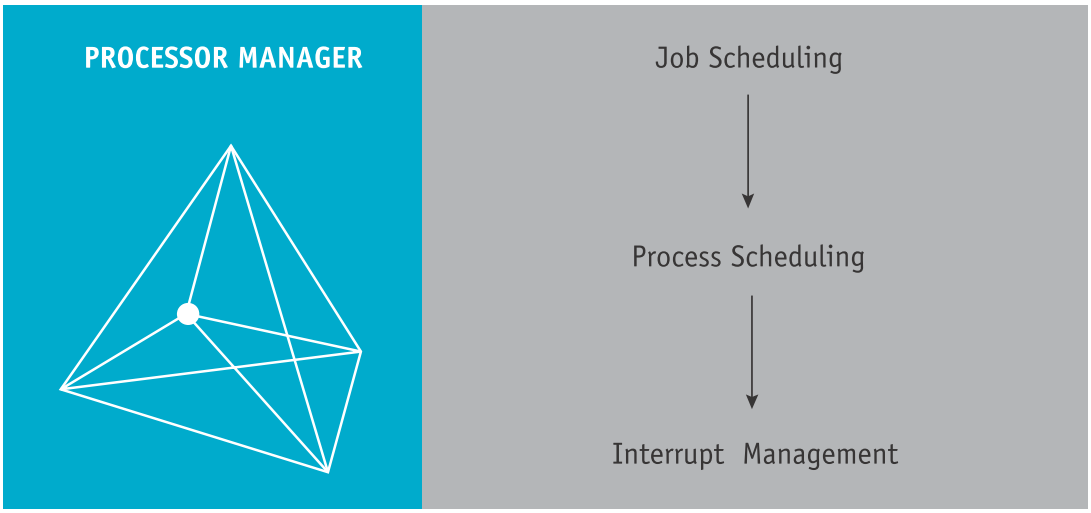
Chapter 4 Processor Management



"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

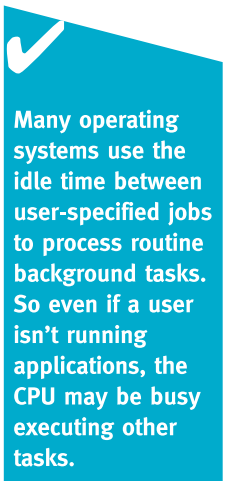
After completing this chapter, you should be able to describe:

- The relationship between job scheduling and process scheduling
- The advantages and disadvantages of several process scheduling algorithms
- The goals of process scheduling policies using a single-core CPU
- The similarities and differences between processes and threads
- The role of internal interrupts and of the interrupt handler

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

Overview

In a simple system, one with a single user and one processor, the processor is busy only when it is executing the user's jobs or system software. 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 show in this chapter, which is devoted to single processor systems. Those with multiple processors and multicore systems are discussed in Chapter 6.



Definitions

Before we begin, let's define some terms. The **processor**, the CPU, is the part of the hardware that performs calculations and executes programs.

A **program** is an inactive unit, such as a file stored on a disk. A program is not a process. For our discussion, 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, sometimes known as a **task**, is a single instance of a program in execution.

A **thread** is created by a process, and it can be scheduled and executed independently of its parent process. A process can consist of multiple threads. In a threaded environment, the process owns the resources that are allocated; it then becomes a more passive element, so its threads become the elements that use resources (such as the CPU). Manipulating threads is less time consuming than manipulating processes, which are more complex. Some operating systems support multiple threads with a single process, while others support multiple processes with multiple threads.

Multithreading allows applications to manage a separate process with several threads of control. Web browsers use multithreading routinely. For instance, one thread can retrieve images while another sends and retrieves e-mail. Multithreading can also increase responsiveness in a time-sharing system, increase resource sharing, and decrease overhead.

Here's a simplified example. If your single-core system allows its 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 (a more common circumstance), then you can request several pictures at the same time, and the browser will set up multiple connections and download several pictures, seemingly at once.

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 assemble a bicycle (perhaps 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 bike.

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 bicycle. Now you find yourself engaged in a very different task: following the instructions in a first-aid kit and using antiseptic and bandages.

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 your bike 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 bike and the second was to bandage the injury. Each step in assembling the bike (Job A) can be called a *process*. The call for help was an *interrupt*; when you left the bike 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 when each job was completed, each was *finished* or terminated.

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

`Get the input for Job A`	(find and read the instructions in the box)
`Identify the resources`	(collect the necessary tools)
`Execute the process`	(follow each step in turn)
`Receive the interrupt`	(receive call for help)
`Perform a context switch to Job B`	(mark your place in the assembly instructions)
`Get the input for Job B`	(find your first-aid kit)
`Identify the resources`	(identify the medical supplies)
`Execute the process`	(follow each first aid step)
`Terminate Job B`	(return home)
`Perform 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 bike)

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 (assembling the bicycle) 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 in this chapter.

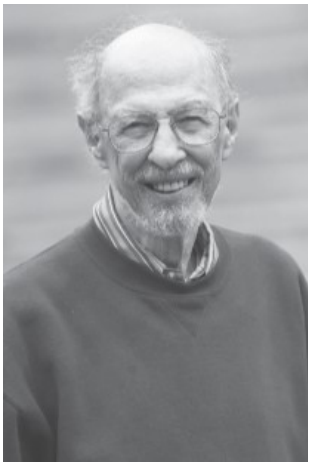
About Multi-Core Technologies

A dual-core, quad-core, or other multi-core CPU has more than one processing element (sometimes 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 such an arrangement helped increase system performance dramatically, the close proximity of these transistors also caused the unintended loss of electrical current and excessive heat that can result in circuit failure.

One solution was to create a single chip (one piece of silicon) that housed two or more processor cores. In other words, a single large processor was replaced with two smaller processors (dual core), or four even smaller processors (quad core). The combined multi-core chips are of approximately the same size as a single-processor chip but produce less current leakage and heat. They also permit multiple calculations to take place at the same time.

Fernando J. Corbato (1926–)

Fernando Corbato is widely recognized for his contributions to the design and development of two operating systems during the 1960s: the Compatible Time-Sharing System (CTSS) and the Multiplexed Information and Computing Service (Multics), both state-of-the-art, multi-user systems. (Years later, Multics became the basis of another operating system called UNIX.) Corbato has won many honors, including the IEEE Computer Society Computer Pioneer Award (1982). In 1976 he was elected to the National Academy of Engineering. In 1982 he became a Fellow of the American Association for the Advancement of Science.



For more information, see *http://amturing.acm.org/award_winners/corbato_1009471.cfm*

Corbato received the ACM 1990 A.M. Turing Award "for his pioneering work in organizing the concepts and leading the development of the general-purpose, large-scale, time-sharing and resource-sharing computer systems, CTSS and Multics."

Jason Dorfman MIT/CSAIL

Multiple core systems are more complex for the Processor Manager to manage than a single core. We discuss these challenges in Chapter 6.

Scheduling Submanagers

The Processor Manager is a composite of at least 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 bike and to bandage the injury, was on a priority basis. Each job was initiated by the Job Scheduler based on certain criteria. Once a job was selected for execution, the Process Scheduler determined when

each step, or set of steps, was executed—a decision that was also based on certain criteria. When you started assembling the bike, each step in the assembly instructions was selected for execution by the Process Scheduler.

The same concepts apply to computer systems, where 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 concerned only with selecting jobs from a queue of incoming jobs and placing them in the process queue based on each job's characteristics. The Job Scheduler's goal is to put the jobs (as they still reside on the disk) in a sequence that best meets the designers or administrator's goals, such as using the system's resources as efficiently 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 requests for input and output (often abbreviated as I/O), then the I/O devices would be kept very busy. The CPU might be busy handling the I/O requests (if an I/O controller were not used), resulting in the completion of very little computation. On the other hand, if the Job Scheduler selected several consecutive jobs with a great deal of computation requirements, then the CPU would be very busy doing that, forcing the I/O devices to remain idle waiting for requests. Therefore, a major goal of the Job Scheduler is to create an order for the incoming jobs that has a balanced mix of I/O interaction and computation requirements, thus balancing the system's resources. As you might expect, the Job Scheduler's goal is to keep most components of the computer system busy most of the time.

Process Scheduler

After a job has been accepted by the Job Scheduler to run, the Process Scheduler takes over that job (and if the operating systems support threads, the Process Scheduler takes responsibility for that function, too). The Process Scheduler determines which processes will get the CPU, when, and for how long. It also decides what to do when processing is interrupted; it determines which waiting lines (queues) the job should be moved to during its execution; and it recognizes when a job has concluded and should be terminated.

The Process Scheduler is a **low-level scheduler** that assigns the CPU to execute the individual actions for those jobs placed on the READY queue by the Job Scheduler. This becomes crucial 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 (often the first I/O cycle) and printing (often the last I/O cycle) are usually brief compared to the time it takes to do the calculations (the CPU cycle).

```
Ask Clerk for the first number
Retrieve the number that's entered (Input #1)
  on the keyboard
Ask Clerk for the second number
Retrieve the second number entered (Input #2)
  on the keyboard
```

Brief I/O Cycle

```
Add the two numbers (Input #1 + Input #2)
Divide the sum from the previous calculation
  and divide by 2 to get the average
  (Input #1 + Input #2) / 2
Multiply the result of the previous calculation
  by 3 to get the average for the quarter
Multiply the result of the previous calculation
  by 4 to get the average for the year
```

CPU Cycle

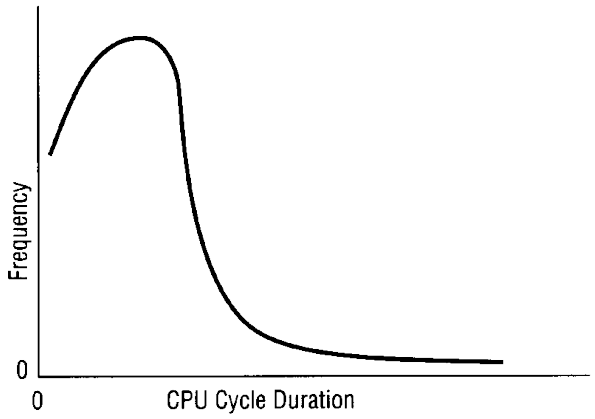
```
Print the average for the quarter (Output #1)
Print the average for the year (Output #2)
```

Brief I/O Cycle

```
End
```

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. Two of these are **I/O-bound** jobs (such as printing a series of documents) that have long I/O cycles and brief CPU cycles and **CPU-bound** jobs (such as finding the first 300,000 prime numbers) that 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 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



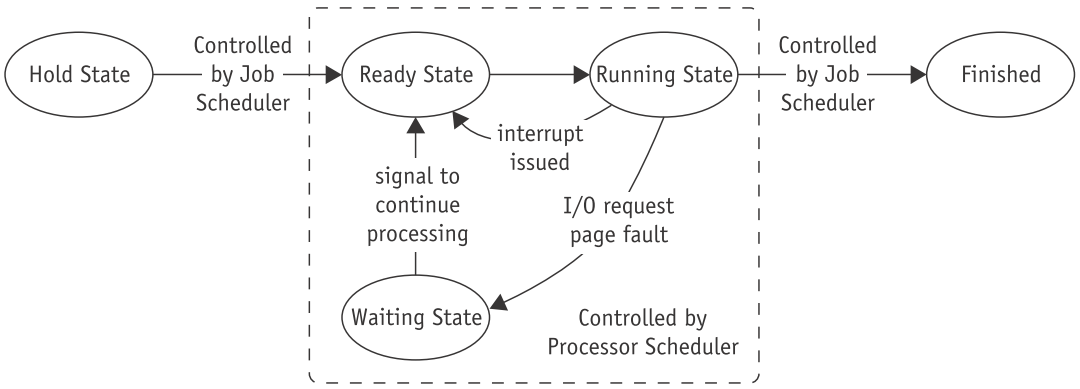
(figure 4.1)

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.

memory to reduce the degree of multiprogramming, which allows other jobs to be completed faster. The jobs that are swapped out and eventually swapped back in are managed by the middle-level scheduler.

Job and Process States

As a job, a process, or a thread moves through the system, its status changes, often from HOLD, to READY, to RUNNING, to WAITING, and eventually to FINISHED, as shown in Figure 4.2. These are called the **job status**, **process status**, or **thread status**, respectively.



(figure 4.2)

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

Here's how a job status can change when a user submits a job to the system. 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.

The job moves to READY after the interrupts have been resolved. 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, and then moves back to READY. Upon completion, the job is FINISHED and returned to the user.

The transition from one job status, job state, to another is initiated by the Job Scheduler, and the transition from one process or thread state to another is initiated by the Process Scheduler. Here's a simple example:

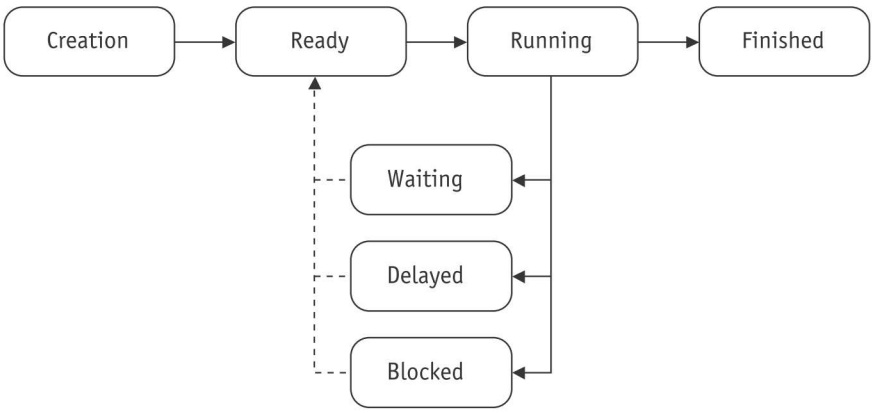
- The job transition from HOLD to READY is initiated by the Job Scheduler (according to a policy that's predefined by the operating system designers). At this point, the availability of enough main memory and any requested devices is checked.
- The transition from READY to RUNNING is handled by the Process Scheduler according to a predefined algorithm (this is discussed shortly).

In a multiprogramming system, the CPU is often be allocated to many jobs, each with numerous processes, making processor management quite complicated.

- The transition from RUNNING back to READY is handled by the Process Scheduler according to a predefined time limit or other criterion, such as a priority interrupt.
- The transition from RUNNING to WAITING is handled by the Process Scheduler and is initiated in response to an instruction in the job such as a command to READ, WRITE, or other I/O request.
- 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 back in the READY queue.
- Eventually, the transition from RUNNING to FINISHED is initiated by the Process Scheduler or the Job Scheduler 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 must be terminated prematurely.

Thread States

As a thread moves through the system it is in one of five states, not counting its creation and finished states, as shown in Figure 4.3. When an application creates a thread, it is made ready by allocating to it the needed resources and placing it in the READY queue. The thread state changes from READY to RUNNING when the Process Scheduler assigns it to a processor. In this chapter we consider systems with only one processor. See Chapter 6 for systems with multiple processors.



(figure 4.3)

A typical thread changes states from READY to FINISHED as it moves through the system.

A thread transitions from RUNNING to WAITING when it has to wait for an event outside its control to occur. For example, a mouse click can be the trigger event for a thread to change states, causing a transition from WAITING to READY. Alternatively, another thread, having completed its task, can send a signal indicating that the waiting thread can continue to execute.

When an application has the capability of delaying the processing of a thread by a specified amount of time, it causes the thread to transition from RUNNING to DELAYED.

When the prescribed time has elapsed, the thread transitions from DELAYED to READY. For example, when using a word processor, the thread that periodically saves a current document can be delayed for a period of time after it has completed the save. After the time has expired, it performs the next save and then is delayed again. If the delay was not built into the application, this thread would be forced into a loop that would continuously test to see if it is time to do a save, wasting processor time and reducing system performance.

A thread transitions from RUNNING to BLOCKED when an I/O request is issued. After the I/O is completed, the thread returns to the READY state. When a thread transitions from RUNNING to FINISHED, all of its resources are released; it then exits the system or is terminated and ceases to exist.

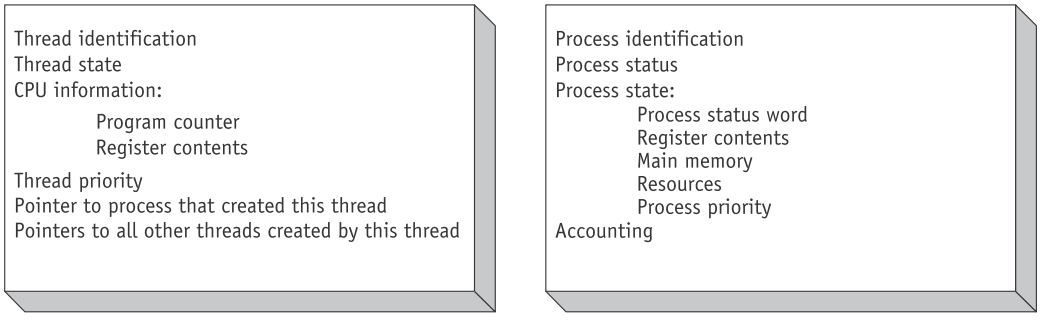
As you can see, the same operations are performed on both traditional processes and threads. Therefore, the operating system must be able to support:

- Creating new threads
- Setting up a thread so it is ready to execute
- Delaying, or putting to sleep, threads for a specified amount of time
- Blocking, or suspending, threads that are waiting for I/O to be completed
- Setting threads to a WAIT state until a specific event has occurred
- Scheduling threads for execution
- Synchronizing thread execution using semaphores, events, or conditional variables
- Terminating a thread and releasing its resources

To do so, the operating system needs to track the critical information for each thread.

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. Similarly, each thread is represented by a similar data structure called a **Thread Control Block** (TCB). Both kinds of control blocks (illustrated in Figure 4.4) contain the basic information that is detailed in Tables 4.1 and 4.2.



(figure 4.4)

Comparison of a typical Thread Control Block (TCB) vs. a Process Control Block (PCB).

Process Identification	Unique identification provided by the Job Scheduler when the job first enters the system and is placed on HOLD.
Process Status	Indicates the current status of the job—HOLD, READY, RUNNING, or WAITING—and the resources responsible for that status.
Process State	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 linking 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. • Elapsed 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, utilization of channels, control units, and devices. • Time spent waiting for I/O completion. • Amount of data read and the number of output records written.

(table 4.1)

Typical contents of a Process Control Block

(table 4.2)

Typical contents of a Thread Control Block

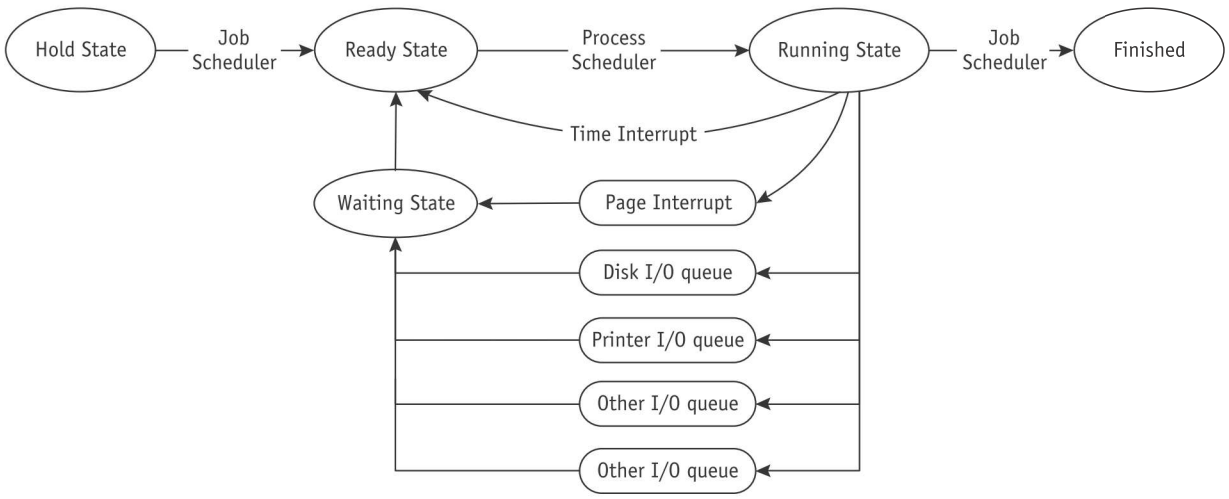
Thread Identification	Unique identification provided by the Process Scheduler when the when the thread is created.
Thread State	The current state of the thread (READY, RUNNING, and so on), which changes as the thread progresses through its execution.
CPU Information	Contains everything that the operating system needs to know about how far the thread has executed, which instruction is currently being performed, and what data is being used.
Thread Priority	Used to indicate the weight of this thread relative to other threads and used to determine which thread should be selected from the READY queue.
Process Pointer	This indicates the process that created the thread.
Sub-thread Pointers	This indicates other subthreads that were created by this thread.

Control Blocks and Queuing

The Process Control Block 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. Likewise, the Thread Control Block is created by the Process Scheduler to track the thread's progress from beginning to end.

Queues use these control blocks to track jobs the same way customs officials use passports to track international visitors. These control blocks contain all of the data 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 control block.

It is the control block, and not the actual process or job, that is linked to other control blocks to form the queues as shown in Figure 4.5, which shows the path of a PCB through execution. Although each control block is not drawn in detail, the reader should imagine each queue as a linked list of PCBs.



(figure 4.5)

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

For example, if we watch a PCB move through its queues, the PCB for each ready job is linked on the READY queue, and 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. Therefore, 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.

Whether they are linking PCBs or TCBs, these queues need to be managed in an orderly fashion, and that's determined by the process scheduling policies and algorithms.

Scheduling Policies

In a multiprogramming environment, there are usually more objects (jobs, processes, threads) 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:

- There are a finite number of resources.
- Some resources, once they're allocated, can't be shared with another job.
- Some resources require operator intervention—that is, they can't be reassigned automatically from job to job.

What's a good **scheduling policy**? Several goals 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 all other 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 are put into groups so they run more efficiently than interactive jobs).
- *Minimize waiting time*. Move jobs out of the READY queue as quickly as possible. This could be done only 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 it doesn't efficiently use its resources. The final decision rests with the system designer or administrator, who must determine which criteria are most important for that specific system. For example, you might want 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." You would select the scheduling policy that most closely satisfies these goals.

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 is 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. In the case of an infinite loop, the process can be stopped and moved to the FINISHED queue whether the policy is preemptive or nonpreemptive.

Scheduling Algorithms

The Process Scheduler relies on a **scheduling algorithm**, based on a specific scheduling policy, to allocate the CPU in the best way to move jobs through the system efficiently. Most systems place an emphasis on fast user **response time**.

To keep this discussion simple, we refer to these algorithms as **process scheduling algorithms**, though they are also used to schedule threads. Here are several algorithms that have been used extensively for these purposes.

First-Come, First-Served

First-come, first-served (**FCFS**) is a nonpreemptive scheduling algorithm that handles all incoming objects 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 First In, First Out (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 READY queue after the jobs before it

runs to completion and the processor becomes available—that is, after all of the jobs before it in the queue have run to completion.

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.

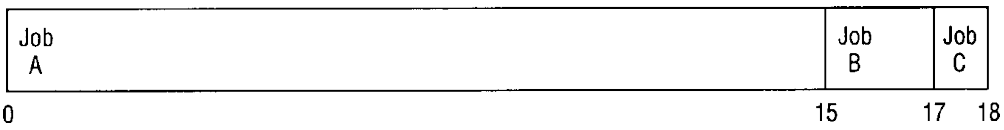
The following examples presume a strictly FCFS environment (no multiprogramming). **Turnaround time** (the time required to execute a job and return the output to the user) is unpredictable with the FCFS policy. For example, consider the following three jobs:

- 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. The timeline shown in Figure 4.6 shows an FCFS algorithm with an arrival sequence of A, B, C.

(figure 4.6)

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



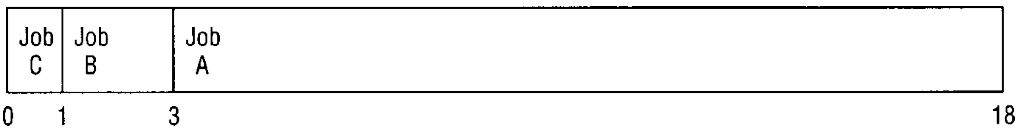
If all three jobs arrive almost simultaneously (at Time 0), we can calculate that the turnaround time (the job's finish time minus arrival time) for Job A is 15, for Job B is 17, and for Job C is 18. So the average turnaround time is:

$$\frac{(15 - 0) + (17 - 0) + (18 - 0)}{3} = 16.67$$

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 4.7.

(figure 4.7)

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

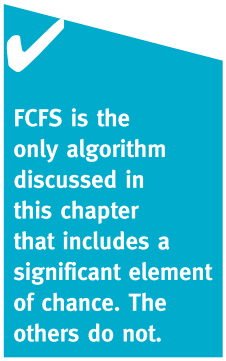


In this example, the turnaround time for Job A is 18, for Job B is 3, and for Job C is 1 and the average turnaround time (shown here in the order in which they finish: Job C, Job B, Job A) is:

$$\frac{(1 - 0) + (3 - 0) + (18 - 0)}{3} = 7.3$$

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.

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 remain stable while the READY list grows (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 an environment that doesn't support multiprogramming.

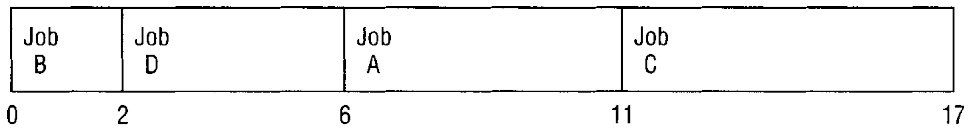
Shortest Job Next

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 possible 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 most interactive systems because users don't estimate in advance the CPU time required to run their jobs.

For example, here are four jobs, all in the READY queue at Time 0. 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 4.8.



(figure 4.8)

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

The average turnaround time (shown here in the order in which they finish: Job B, Job D, Job A, Job C) is:

$$\frac{(2 - 0) + (6 - 0) + (11 - 0) + (17 - 0)}{4} = 9.0$$

Let's take a minute to see why this algorithm consistently gives the minimum average turnaround time. We use the previous example to derive a general formula.

We can see in Figure 4.7 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.

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

Priority Scheduling

Priority scheduling is one of the most common scheduling algorithms for batch systems and is a nonpreemptive algorithm (in a batch environment). 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 are discussed later in this chapter in the section on multiple queues.

Shortest Remaining Time

Shortest remaining time (**SRT**) is a preemptive version of the SJN algorithm. The processor is allocated to the job closest to completion—but even this job can be interrupted 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 can work well in batch environments, because it can give preference to short jobs. A disadvantage is that SRT involves more overhead than SJN—it requires the operating system to frequently monitor the CPU time for all the jobs in the READY queue and it must perform context

When using SRT, if several jobs have the same amount of time remaining, the job that has been waiting the longest goes next. It uses the FCFS algorithm to break the tie.

switching for the jobs being swapped at preemption time (not necessarily swapped out to the disk, although this might occur as well).

The example in Figure 4.9 shows how the SRT algorithm works with four jobs that arrived in quick succession (one CPU cycle apart).

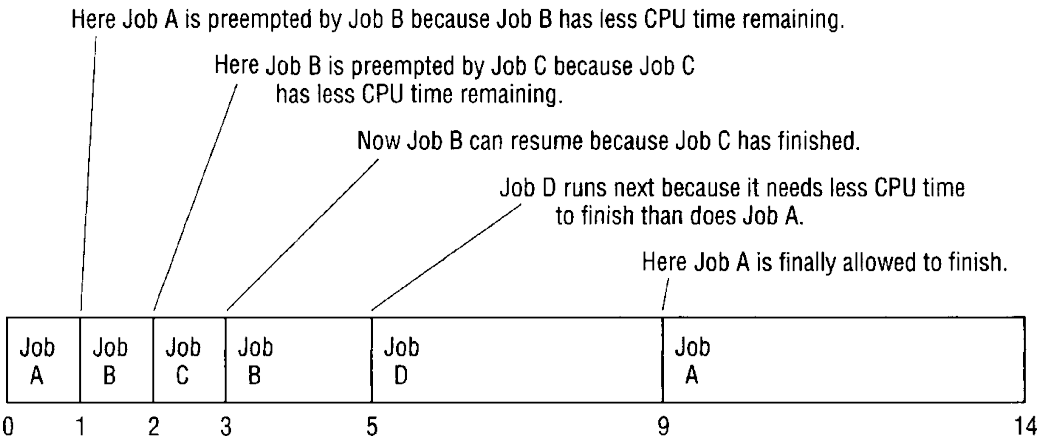
Arrival time:	0	1	2	3
Job:	A	B	C	D
CPU cycle:	6	3	1	4

The turnaround time for each job is its completion time minus its arrival time. The resulting turnaround time for each job is:

Job:	A	B	C	D
Completion Time minus Arrival Time	14-0	5-1	3-2	9-3
Turnaround	14	4	1	6

So the average turnaround time is:

$$\frac{(14 - 0) + (5 - 1) + (3 - 2) + (9 - 3)}{4} = 6.25$$



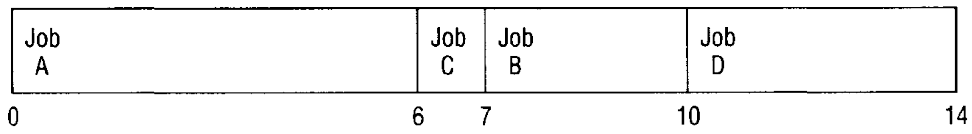
(figure 4.9) *Timeline for job sequence A, B, C, D using the preemptive SRT algorithm. Each job is interrupted after one CPU cycle if another job is waiting with less CPU time remaining.*

How does that compare to the same problem using the nonpreemptive SJN policy? Figure 4.10 shows the same situation with the same arrival times using SJN. In this case, the turnaround time is:

Job:	A	B	C	D
Completion Time minus Arrival Time	6-0	10-1	7-2	14-3
Turnaround:	6	9	5	11

So the average turnaround time is:

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



(figure 4.10)

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

Note in Figure 4.10 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, in 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 the saving of 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 is required by all preemptive algorithms so jobs can pick up later where they left off. 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 and contributes to overhead. 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 the computing version of two children taking turns using the television remote control. Round Robin is easy to implement. It 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**; its size is crucial to the performance of the system. It can vary from 100 milliseconds to 1 or 2 seconds.

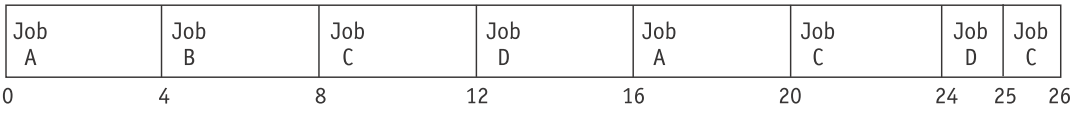
Jobs are placed in the READY queue using a first-come, first-served scheme. The Process Scheduler selects the first job from the front of the queue, sets the timer to the time quantum, and allocates the CPU to this job. If processing isn't finished when time expires, the job is preempted and put at the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

With Round Robin and a queue with numerous processes, each process gets its first access to the processor before the first process gets access a second time.

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 4.11 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 4.11) *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
Completion Time minus Arrival Time	20-0	8-1	26-2	25-3
Turnaround:	20	7	24	22

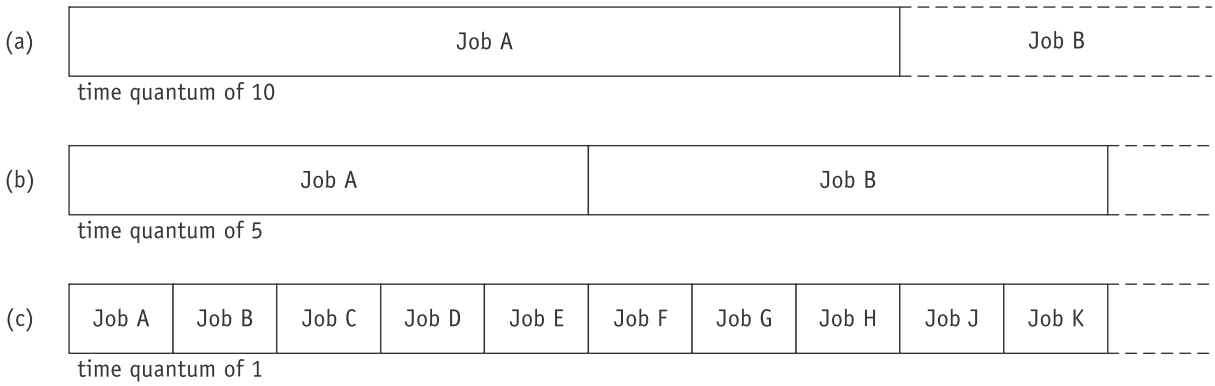
So the average turnaround time is:

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

Note that in Figure 4.11, 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.

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 4.12 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 4.12, 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 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.



(figure 4.12)

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.

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 a time-critical environment, the system is expected to respond immediately. If it's an archival system, turnaround time and overhead become critically 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.

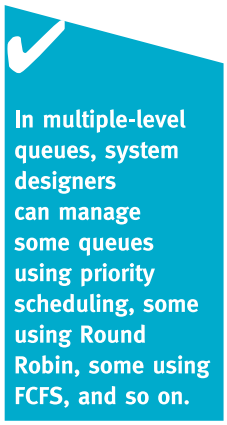
Multiple-Level Queues

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 a different queue 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 can be 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. With multiple-level queues, the system designers can choose to use different algorithms for different queues, allowing them to combine the advantages of several algorithms. For example, within each queue, the jobs are served in FCFS fashion or use some other scheme instead.



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). We explore each of these methods with the following four cases.

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, such as 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. 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. 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 causing it to never be processed. Indefinite postponement is a major problem when allocating resources and one that is discussed in detail in Chapter 5.

Earliest Deadline First

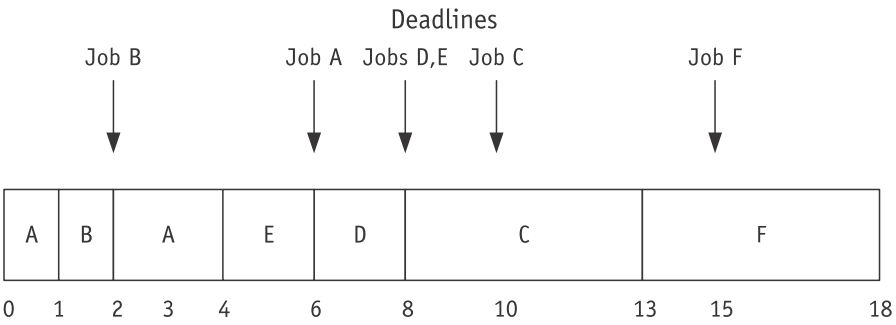
Earliest Deadline First (EDF), known as a dynamic priority algorithm, is a preemptive scheduling algorithm built to address the critical processing requirements of real-time systems and their pressing deadlines. Contrary to the fixed priority scheme explored earlier, where the priority of a job does not change after it enters the system, with EDF the priority can be adjusted as it moves through execution from START to FINISHED.

The primary goal of EDF is to process all jobs in the order that is most likely to allow each to run to completion before reaching their respective deadlines. Initially, the priority assigned to each job is based on the amount of time remaining until the job's impending deadline—and that priority is inversely proportional to its absolute deadline. So, in its simplest sense: the closer the deadline, the higher the priority. Sometime two or more jobs share the same deadline, in which case the tie is broken using any scheme such as first in, first out. Remember, the goal is to complete all jobs before each one reaches its deadline.

With this algorithm, a job's deadline can change and the algorithm can change its priority accordingly. For example, consider the following job stream and the timeline shown in Figure 4-13.

Job:	A	B	C	D	E	F
Arrival time:	0	1	2	3	3	5
Execution Time:	3	1	5	2	2	5
Deadline:	6	2	10	8	8	15
Time-before-deadline (at arrival time)	6	1	8	5	5	10

The tie between Jobs D and E (which both arrived at the same time) was broken arbitrarily with E going first. Notice that if we reorder the job stream by absolute deadline (regardless of the arrival time of each), we can see the order in which they might finish: Job B (at Time 2), Job A (Time 6), Job D and Job E (both at Time 8), Job C (Time 10), and Job F (Time 15).



(figure 4-13)

EDF dynamic priority timeline showing processing order and deadlines, which were met by Jobs A-E.

Using this algorithm, the priority of each job can change as more important jobs enter the system for processing, and the job with the closest deadline immediately assumes highest priority. (Job A in Figure 4-13 assumed highest priority twice before it ran to completion.)

This example illustrates one of the difficulties with the EDF algorithm. By adding the total amount of computation/execution time, we can see that a total of 18 units will be required (if we ignore overhead operations), and yet the deadline for these six jobs is at Time 15. It's not possible to perform 18 units of execution in 15 units of time.

Additional problems can occur because of the algorithm's dynamic nature. For example, it is impossible to predict job throughput because the priority of a given job rises or falls depending on the mix of other jobs in the system. A job with a closer deadline will cause the other waiting jobs to be delayed. The EDF algorithm also has high

overhead requirements because it constantly evaluates the deadlines of all the jobs awaiting execution.

Beware that while EDF is written to meet the needs of real-time environments, there is no guarantee that it is possible for it to succeed, and deadlines can still be missed when using this algorithm. Still, of the schemes discussed in this chapter, it is designed well to deal with the stringent requirements of dynamic real-time computing systems.

Managing 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, such as the one that occurs when the time quantum expires and the processor is deallocated from the running job and allocated to another one.

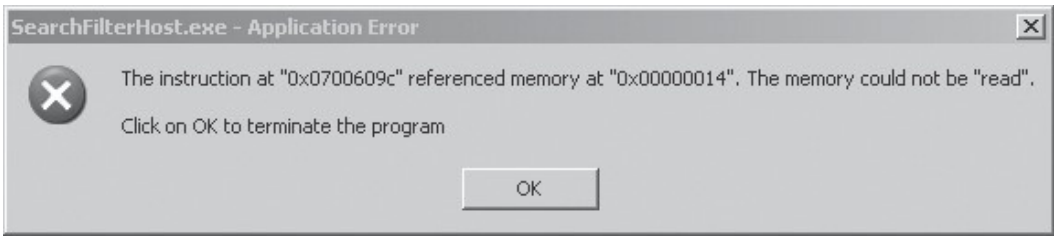
There are other interrupts that are caused by events internal to the process. I/O interrupts are issued when a READ or WRITE command is issued (and we detail those in Chapter 7). Internal interrupts, or synchronous interrupts, also occur as a direct result of the arithmetic operation or other instruction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

For example, interrupts can be generated by illegal arithmetic operations, including the following:

- Attempts to divide by zero (called an exception)
- 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, such as the one shown in Figure 4.14
- Attempts to use an undefined operation code
- Operating on invalid data
- Unauthorized attempts to make system changes, such as trying to change the size of the time quantum



(figure 4.14) Sample Windows screen showing that the interrupt handler has stepped in after an invalid operation.

The control program that handles the interruption sequence of events is called the **interrupt handler**. When the operating system detects an error that is not recoverable, 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 the 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 fatal, nonrecoverable interrupts, 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 suspends 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 quantum interrupt, the job process or thread is already on the READY queue.

Conclusion

The Processor Manager must allocate the CPU among all the system's users and all of their jobs, processes, and threads. In this chapter we've made the distinction between job scheduling (the selection of incoming jobs based on their characteristics) and process and thread 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 the strengths and weaknesses of each one that's available in the context of the system's requirements. Table 4.3 shows how the algorithms presented in this chapter compare.

In the next chapter we explore the demands placed on the Processor Manager as it attempts to synchronize execution of every job admitted to the system to avoid deadlocks, livelock, and starvation.

(table 4.3)

Comparison of the scheduling algorithms discussed in this chapter

Algorithm	Policy Type	Disadvantages	Advantages
First Come, First Served	Nonpreemptive	Unpredictable turnaround times; has an element of chance	Easy to implement
Shortest Job Next	Nonpreemptive	Indefinite postponement of some jobs; requires execution times in advance	Minimizes average waiting time
Priority Scheduling	Nonpreemptive	Indefinite postponement of some jobs	Ensures fast completion of important jobs
Shortest Remaining Time	Preemptive	Overhead incurred by context switching	Ensures fast completion of short jobs
Round Robin	Preemptive	Requires selection of good time quantum	Provides reasonable response times to interactive users; provides fair CPU allocation
Multiple-Level Queues	Preemptive/ Nonpreemptive	Overhead incurred by monitoring queues	Flexible scheme; allows aging or other queue movement to counteract indefinite postponement; is fair to CPU-bound jobs
Earliest Deadline First	Preemptive	Overhead required to monitor dynamic deadlines	Attempts timely completion of jobs

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 computation before issuing an I/O request. It contrasts with *I/O-bound*.

earliest deadline first (EDF): a preemptive process scheduling policy (or algorithm) that selects processes based on the proximity of their deadlines (appropriate for real-time environments).

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 substantial free time for the CPU. It contrasts with *CPU-bound*.

indefinite postponement: signifies that a job's execution is delayed indefinitely.

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 certain 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 state 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: 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: a low-level scheduler that establishes the order in which processes in the READY queue will be served by the CPU.

process status: information stored in the job's PCB that indicates the current location in memory 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: a unit of instructions.

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

response time: one measure of the efficiency of an interactive system that tracks the time required for the system to 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.

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

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

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 process that can run independently. Multithreaded systems can have several threads running at one time with the same or different priorities.

Thread Control Block (TCB): a data structure that contains information about the current status and characteristics of a thread.

thread status: information stored in the thread control block that indicates the current position of the thread and the resources responsible for that status.

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

For more background on a few of the topics discussed in this chapter, begin a search with these terms:

- Thread Scheduling Priorities
- CPU Cycle Time
- Processor Bottleneck
- Processor Queue Length
- I/O Interrupts

Exercises

Research Topics

A. Multi-core technology can often, but not 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 laptop 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. Compare and contrast a process and a thread.
2. Which scheduler is responsible for scheduling threads in a multithreading system?
3. Five jobs arrive nearly simultaneously for processing and their estimated CPU cycles are, respectively: Job A = 12, Job B = 2, Job C = 15, Job D = 7, and Job E = 3 ms.
   a. Using FCFS, and assuming the difference in arrival time is negligible, in what order would they be processed? What is the total time required to process all five jobs? What is the average turnaround time for all five jobs?

b. Using SJN, and assuming the difference in arrival time is negligible, in what order would they be processed? What is the total time required to process all five jobs? What is the average turnaround time for all five jobs?

4. Assume a multiple level queue system with a variable time quantum per queue, and that the incoming job needs 50 ms to run to completion. If the first queue has a time quantum of 5 ms 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? Explain how much time it spends in each queue.
5. Using the same multiple level queue system from the previous exercises, if a job needs 130 ms to run to completion, how many times will the job be interrupted and on which queue will it finish its execution? Does it matter if there are other jobs in the system?
6. Assume that your system has one queue for jobs waiting for printing and another queue for those waiting for access to a disk. Which queue would you expect to have the faster response? Explain your reasoning.
7. Using SJN, calculate the start time and finish time for each of these seven jobs:

Job	Arrival Time	CPU Cycle
A	0	2
B	1	11
C	2	4
D	4	1
E	5	9
F	7	4
G	8	2

8. Given the following information:

Job	Arrival Time	CPU Cycle
A	0	15
B	2	2
C	3	14
D	6	10
E	9	1

Calculate which jobs will have arrived ready for processing by the time the first job is finished or first interrupted using each of the following scheduling algorithms.

a. FCFS
b. SJN
c. SRT
d. Round Robin (use a time quantum of 5, but ignore the time required for context switching and natural wait)

9. Using the same information from the previous exercise, calculate the start time and finish time for each of the five jobs using each of the following scheduling algorithms. It may help to draw the timeline.

a. FCFS
b. SJN
c. SRT
d. Round Robin (use a time quantum of 5, but ignore the time required for context switching and natural wait)

10. Using the same information given for Exercise 8, compute the turnaround time for every job for each of the following scheduling algorithms (ignore context switching overhead times). It may help to draw the timeline.

a. FCFS
b. SJN
c. SRT
d. Round Robin (using a time quantum of 5)

11. Given the following information for a real-time system using EDF:

Job:	A	B	C	D	E	F
Arrival time:	0	0	1	1	3	6
Execution Time:	3	1	6	2	7	5
Deadline:	6	1	44	2	16	15
Time-before-deadline (at arrival time)						

Compute the time-before-deadline for each incoming job. Give the order in which the six jobs will finish, and identify any jobs that fail to meet their deadline. It may help to draw a timeline.

Advanced Exercises

12. Consider this variation of Round Robin. 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. 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. Explain the advantages and disadvantages of this scheduling policy? Identify the user group that would find this most advantageous.

13. When using a personal computer, it can be easy to determine when a job is caught in an infinite loop or system-wide freeze. The typical solution to this problem is for the user to manually intervene and terminate the offending job, or in the worst case, all jobs. 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, such as a task that is calculating the first 300,000 prime numbers.
14. Some guidelines for selecting the right time quantum were given in this chapter. As a system designer, which guidelines do you prefer? Which would the average user prefer? How would you know when you have chosen the best time quantum? What factors would make this time quantum best from the system's point of view?
15. 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

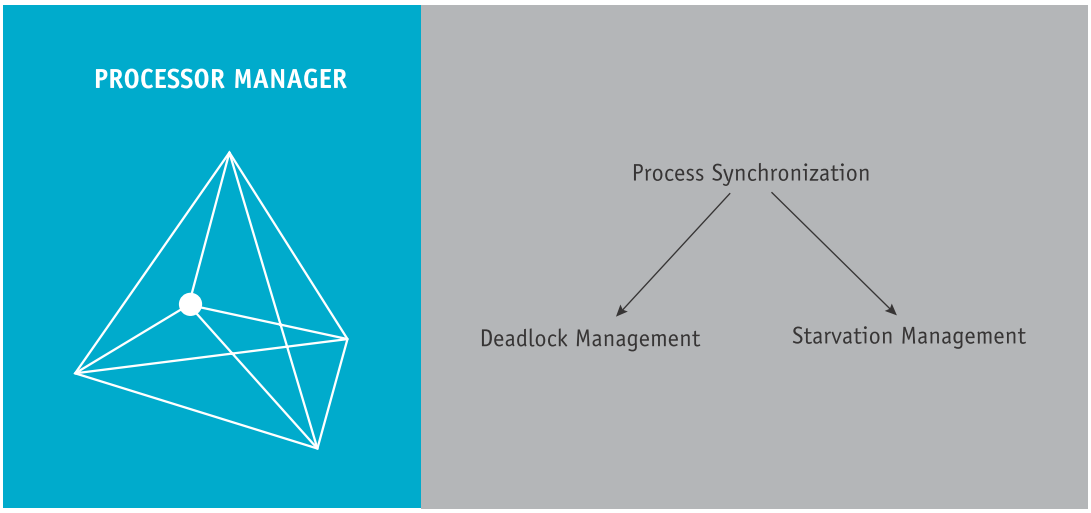
16. Write a program that will simulate FCFS, SJN, SRT, and Round Robin scheduling algorithms. For each algorithm, the program should compute waiting time and turnaround time of every job as well as the average waiting time and average turnaround time. The average values should be consolidated in a table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

Job	Arrival Times	CPU Cycle (in milliseconds)
A	0	16
B	3	2
C	5	11
D	9	6
E	10	1
F	12	9
G	14	4

Job	Arrival Times	CPU Cycle (in milliseconds)
H	16	14
I	17	1
J	19	8

17. Modify your program from Exercise 16 to generate a random job stream and change the context switching time to 0.4 ms. Compare outputs from both runs and discuss which would be the better policy. Describe any drastic changes encountered (or a lack of changes), and explain why.

Chapter 5 Process Management



"We have all heard the story of the animal standing in doubt between two stacks of hay and starving to death."

—Abraham Lincoln *(1809–1865)*

Learning Objectives

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

- The differences among deadlock, race, and starvation
- Several causes of system deadlock and livelock
- The difference between preventing and avoiding deadlocks
- How to detect and recover from deadlocks
- How to detect and recover from starvation
- The concept of a race and how to prevent it

We've already looked at resource sharing from two perspectives—that of sharing memory and sharing one processor—but the processor sharing described thus far was the best case scenario, free of conflicts and complications. In this chapter, we address the problems caused when many processes compete for relatively few resources, causing the system to stop responding as it should and rendering it unable to service all of the necessary processes.

Let's look at how a lack of **process synchronization** can result in two extreme conditions: deadlock or starvation.

System deadlock has been known through the years by many descriptive phrases, including "deadly embrace," "Catch 22," and "blue screen of death," to name a few. A simple example of a deadlock is illustrated in Figure 5.1. Let's say the ice cream store is closing and you just got the last available ice cream sundae—but the next person in line grabbed the only available spoon! If no additional ice cream sundaes AND no additional spoons become available, AND neither one of you decides to give up your resources, then a deadlock has occurred. Notice that it takes a combination of circumstances for a deadlock to occur.

(a) You (b) Someone Else

(figure 5.1)

A very simple example of a deadlock: You hold the ice cream (a) but you need the only available spoon to eat it. Someone else holds the only spoon (b) but has no ice cream to eat.

In computer systems, a deadlock is a system-wide tangle of resource requests that begins when two or more jobs are put on hold, each waiting for a vital resource to become available. The problem builds when the resources needed by those jobs are the resources held by other jobs that are also waiting to run but cannot because they're waiting for other unavailable resources. The tangled jobs come to a standstill. The deadlock is complete if the remainder of the system comes to a standstill as well. When the situation can't be resolved by the operating system, then intervention is required.

Deadlock, Livelock, and Starvation

A **deadlock** is most easily described with an example that we return to throughout the chapter—a narrow staircase in a building. The staircase was built as a fire escape route, but people working in the building often take the stairs instead of waiting for the slow elevators. Traffic on the staircase moves well unless two people, traveling in opposite directions, need to pass on the stairs—there's room for only one person on each step. There's a landing at each floor that is wide enough for two people to share, but the stairs are not—they can be allocated to only one person at a time. In this example, the staircase is the system and the steps and landings are the resources. Problems occur when someone going up the stairs meets someone coming down, and each refuses to retreat to a wider place. This creates a deadlock, which is the subject of much of our discussion on process synchronization.

Similarly, if two people on a landing try to pass each other but cannot do so because as one steps to the right, the other steps to the left, and vice versa, then they will continue moving but neither will ever move forward. This is called **livelock.**

On the other hand, if a few patient people wait on the landing for a break in the opposing traffic, and that break never comes, they could wait there forever. That results in **starvation,** an extreme case of indefinite postponement, and is discussed at the end of this chapter.

Deadlock

Deadlock is more serious than indefinite postponement or starvation because it affects more than one job. Because resources are being tied up, the entire system (not just a few programs) is affected. There's no simple and immediate solution to a deadlock; no one can move forward until someone moves out of the way, but no one can move out of the way until either someone advances or everyone behind moves back. Obviously, it requires outside intervention. Only then can the deadlock be resolved.

Deadlocks became prevalent with the introduction of interactive systems, which generally improve the use of resources through dynamic resource sharing, but this capability also increases the possibility of deadlocks.

In some computer systems, deadlocks are regarded as a mere inconvenience that causes delays. But for real-time systems, deadlocks cause critical situations. For example, a deadlock in a hospital's life support system or in the guidance system aboard an aircraft could endanger lives. Regardless of the environment, the operating system must either prevent deadlocks or resolve them when they happen. (In Chapter 12, we learn how to calculate system reliability and availability, which can be affected by processor conflicts.)

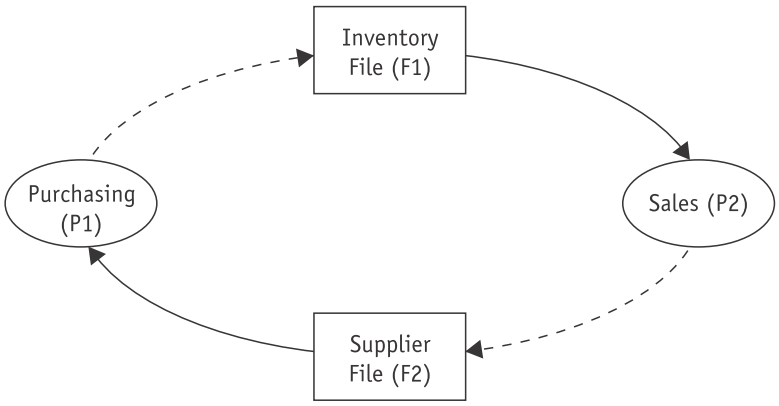
Seven Cases of Deadlock or Livelock

A deadlock usually occurs when nonsharable, nonpreemptable resources, such as files, printers, or scanners, are allocated to jobs that eventually require other nonsharable, nonpreemptive resources—resources that have been locked by other jobs. However, deadlocks aren't restricted to files, printers, and scanners. They can also occur on sharable resources that are locked, such as disks and databases.

Directed graphs visually represent the system's resources and processes and show how they are deadlocked. Using a series of squares for resources, circles for processes, and connectors with arrows for requests, directed graphs can be manipulated to understand how deadlocks occur.

Case 1: Deadlocks on File Requests

If jobs are allowed to request and hold files for the duration of their execution, a deadlock can occur, as the simplified directed graph shown in Figure 5.2 illustrates.



(figure 5.2)

Case 1. These two processes, shown as circles, are each waiting for a resource, shown as rectangles, that has already been allocated to the other process, thus creating a deadlock.

For example, consider the case of a home construction company with two application programs, Purchasing and Sales, which are active at the same time. Both need to access two separate files, called Inventory and Suppliers, to read and write transactions. One day the system deadlocks when the following sequence of events takes place:

1. The Purchasing process accesses the Suppliers file to place an order for more lumber.
2. The Sales process accesses the Inventory file to reserve the parts that will be required to build the home ordered that day.
3. The Purchasing process doesn't release the Suppliers file, but it requests the Inventory file so it can verify the quantity of lumber on hand before placing its order for more. However, Purchasing is blocked because the Inventory file is being held by Sales.

4. Meanwhile, the Sales process doesn't release the Inventory file (because it needs it), but requests the Suppliers file to check the schedule of a subcontractor. At this point, the Sales process is also blocked because the Suppliers file is already being held by the Purchasing process.

In the meantime, any other programs that require the Inventory or Suppliers files will be put on hold as long as this situation continues. This deadlock will remain until one of the two programs is closed or forcibly removed and its file is released. Only then can the other program continue and the system return to normal.

Case 2: Deadlocks in Databases

A deadlock can also occur if two processes access and lock records in a database.

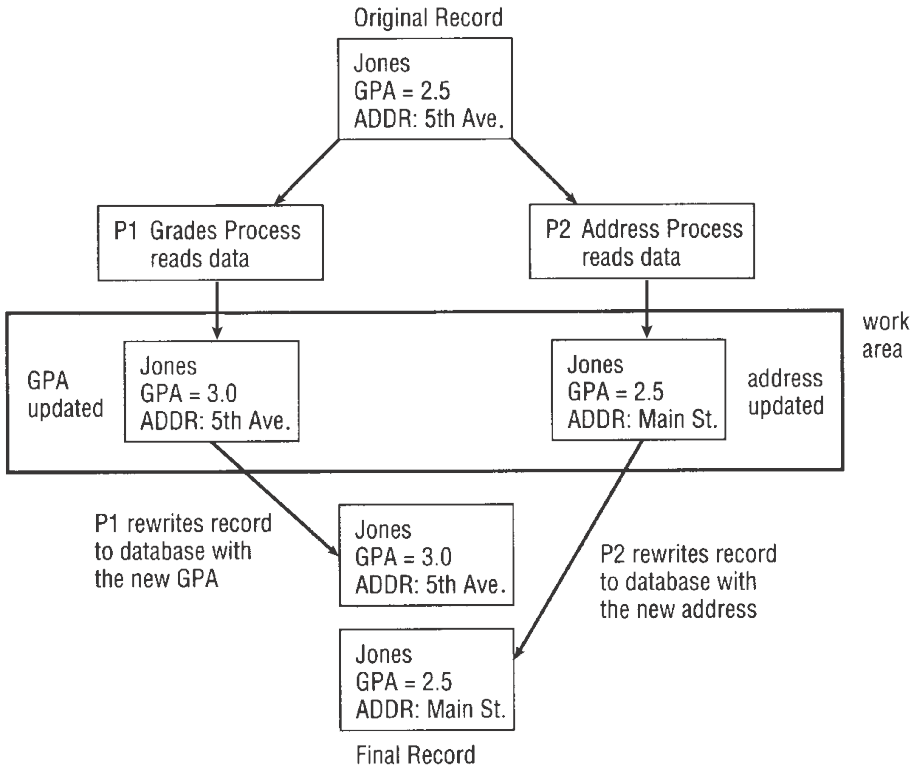
To appreciate the following scenario, remember that database queries and transactions are often relatively brief processes that either search or modify parts of a database. Requests usually arrive at random and may be interleaved arbitrarily.

Database **locking** is a technique used to guarantee the integrity of the data through which the user locks out all other users while working with the database. Locking can be done at three different levels: the entire database can be locked for the duration of the request; a subsection of the database can be locked; or only the individual record can be locked. Locking the entire database (the most extreme and most successful solution) prevents a deadlock from occurring, but it restricts access to the database to one user at a time and, in a multiuser environment, response times are significantly slowed; this is normally an unacceptable solution. When the locking is performed on only one part of the database, access time is improved, but the possibility of a deadlock is increased because different processes sometimes need to work with several parts of the database at the same time.

Here's a system that locks each record in the database when it is accessed and keeps it locked until that process is completed. Let's assume there are two processes (The sales process and address process), each of which needs to update a different record (the final exam record and the current address record), and the following sequence leads to a deadlock:

1. The sales process accesses the first quarter record and locks it.
2. The address process accesses the current address record and locks it.
3. The sales process requests the current address record, which is locked by the address process.
4. The address process requests the first quarter record, which is locked by the sales process.

If locks are *not* used to preserve database integrity, the resulting records in the database might include only some of the data—and their contents would depend on the



(figure 5.3)

Case 2. P1 finishes first and wins the race but its version of the record will soon be overwritten by P2. Regardless of which process wins the race, the final version of the data will be incorrect.

order in which each process finishes its execution. Known as a **race** between processes, this is illustrated in the example shown in Figure 5.3. (Leaving all database records unlocked is an alternative, but that leads to other difficulties.)

Let's say you are a student of a university that maintains most of its files on a database that can be accessed by several different programs, including one for grades and another listing home addresses. You've just moved at the end of fall term, so you send the university a change of address form. It happens to be shortly after grades are submitted. One fateful day, both programs race to access a single record in the database:

1. The grades process (P1) is the first to access your college record (R1), and it copies the record to its own work area.
2. Almost simultaneously, the address process (P2) accesses the same record (R1) and copies it to its own work area. Now the same record is in three different places: the database, the work area for grade changes, and the work area for address changes.
3. The grades process changes your college record by entering your grades for the fall term and calculating your new grade average.
4. The address process changes your college record by updating the address field.
5. The grades process finishes its work first and writes its version of your college record back to the database. Now, your record has updated grades, but your address hasn't changed.

A race introduces the element of chance, an element that's totally unacceptable in database management. The integrity of the database must be upheld.

6. The address process finishes and rewrites its updated version of your record back to the database. This version has the new address, but it replaced the version that contains your grades! At this point, according to the database, you have no grade for this term.

If we reverse the order and say that the address process wins the race, your grades will be updated but not your address. Depending on your success in the classroom, you might prefer one mishap over the other. From the operating system's point of view, both alternatives are unacceptable, because both are incorrect and allow the database to become corrupted. A successful operating system can't allow the integrity of the database to depend on luck or a random sequence of events.

Case 3: Deadlocks in Dedicated Device Allocation

The use of a group of dedicated devices, such as two audio recorders, can also deadlock the system. Remember that dedicated devices cannot be shared.

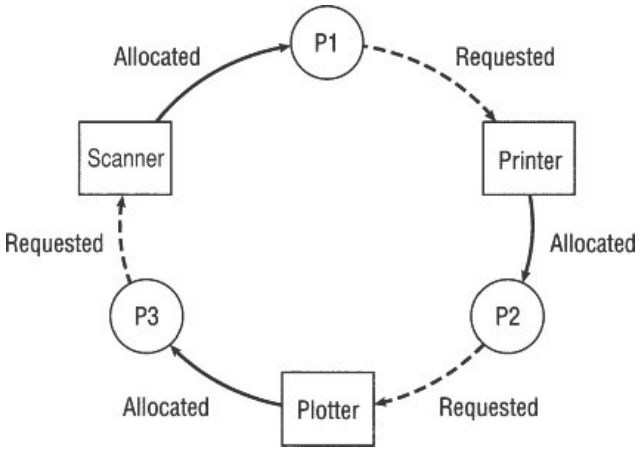
Let's say two administrators, Chris and Jay, from the local board of education are each running education programs and each has a process (P1 and P2, respectively) that will eventually need to use both audio recorders (R1 and R2) together to copy a school board hearing. The two recorders are available at the time when both make the request; the system's allocation policy states that each recorder is to be allocated on an "as requested" basis. Soon the following sequence transpires:

1. Chris's process requests the recorder closest to the office, Recorder #1, and gets it.
2. Jay's process requests the recorder closest to the printer, Recorder #2, and gets it.
3. Chris's process then requests Recorder #2, but is blocked and has to wait.
4. Jay's process requests Recorder #1, but is blocked and the wait begins here, too.

Neither Chris's or Jay's job can continue because each is holding one recorder while waiting for the other process to finish and release its resource—an event that will never occur according to this allocation policy.

Case 4: Deadlocks in Multiple Device Allocation

Deadlocks aren't restricted to processes that are contending for the same type of device; they can happen when several processes request, and hold on to, several dedicated devices while other processes act in a similar manner, as shown in Figure 5.4.



(figure 5.4)

Case 4. Three processes, shown as circles, are each waiting for a device that has already been allocated to another process, thus creating a deadlock.

Consider the case of an engineering design firm with three programs (P1, P2, and P3) and three dedicated devices: scanner, printer, and plotter. The following sequence of events will result in deadlock:

1. Program 1 (P1) requests and gets the only scanner.
2. Program 2 requests and gets the only printer.
3. Program 3 requests and gets the only plotter.
4. Now, Program 1 requests the printer but is blocked.
5. Then, Program 2 requests the plotter but is blocked.
6. Finally, Program 3 requests the scanner but is blocked and the deadlock begins.

As was the case in the earlier examples, none of these programs can continue because each is waiting for a necessary resource that's already being held by another.

Case 5: Deadlocks in Spooling

Although in the previous example the printer was a dedicated device, printers can be sharable devices and join a category called "virtual devices" that uses high-speed storage to transfer data between it and the CPU. The spooler accepts output from several users and acts as a temporary storage area for all output until the printer is ready to accept it. This process is called **spooling**. However, if the printer needs all of a job's output before it will begin printing, but the spooling system fills the available space with only partially completed output, then a deadlock can occur. It happens like this.

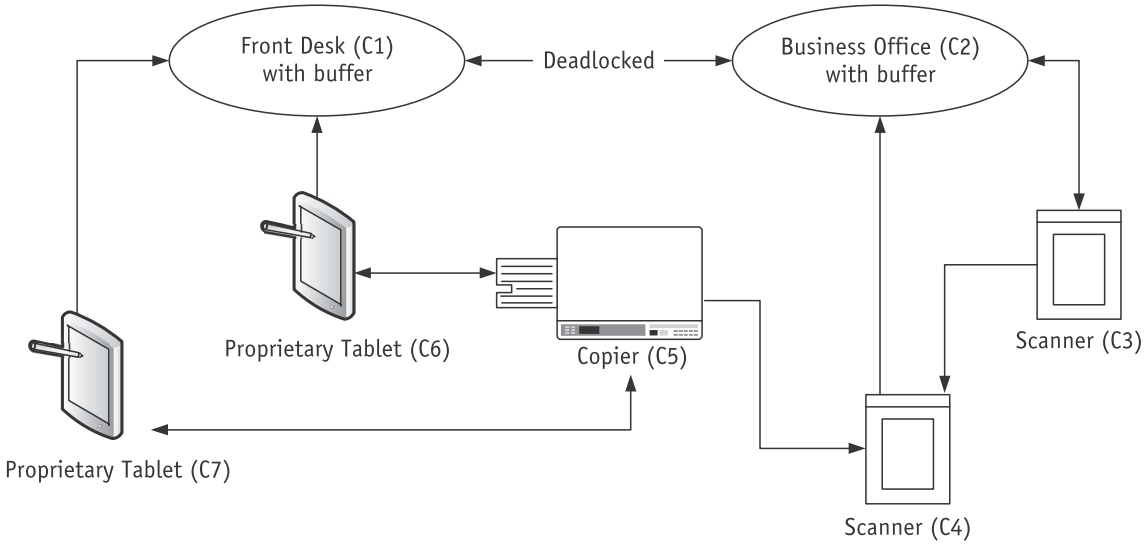
Let's say it's one hour before the big project is due for a computer class. Twenty-six frantic programmers key in their final changes and, with only minutes to spare, all issue print commands. The spooler receives the pages one at a time from each of the students but the pages are received separately, several page ones, page twos, and so on. The printer is ready to print the first completed document it gets, but as the spooler canvasses its files, it has the first page for many programs but the last page for none of them. Alas, the spooler is full of partially completed output—so no other pages can be accepted—but

none of the jobs can be printed (which would release their disk space) because the printer accepts only completed output files. It's an unfortunate state of affairs.

This scenario isn't limited to printers. Any part of the computing system that relies on spooling, such as one that handles incoming jobs or transfers files over a network, is vulnerable to such a deadlock.

Case 6: Deadlocks in a Network

A network that's congested or has filled a large percentage of its I/O buffer space can become deadlocked if it doesn't have protocols to control the flow of messages through the network, as illustrated in Figure 5.5.



(figure 5.5)

Case 6, deadlocked network flow. Notice that only two nodes, C1 and C2, have buffers. Each line represents a communication path. The arrows indicate the direction of data flow.

For example, a medium-sized hotel has seven computing devices on a network, each on different nodes, but only two of those nodes use a buffer to control the stream of incoming data. The front desk (C1) receives messages from the business office (C2) and from two tablets (C6 and C7), and it sends messages to only the business office. All of the communication paths and possible interactions are shown in Figure 5.5.

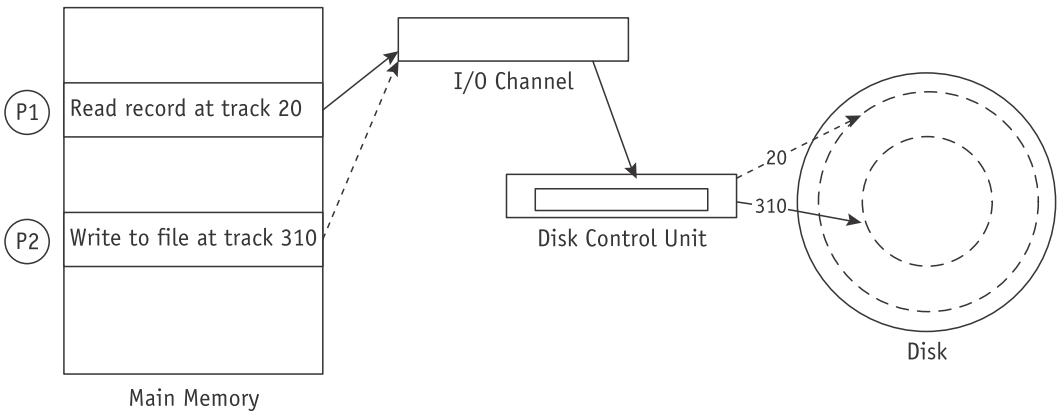
- Messages that are received by front desk from the two tablets and that are destined for the business office are buffered in its output queue until they can be sent.
- Likewise, messages received by the business office from the two scanners and destined for the front desk are buffered in its output queue.
- On a busy day, as data traffic increases, the length of each output queue increases until all of the available buffer space is filled.

- At this point, the front desk can't accept any more messages (from the business office or any other networked device) because there's no more buffer space available to store them.
- For the same reason, the business office can't accept any messages from the front desk or any other computer, not even a request to send data from the buffer, which could help alleviate the congestion.
- The communication path between the front desk and the business office has become deadlocked. Because the front desk can't send messages to any other computer except the one in the business office, and it can only receive (and not send) messages from the two tablets, those routes also become deadlocked.

At this point, the front desk can't alert the business office about the problem without having someone run there with a message (sometimes called a "sneaker net"). This deadlock can't be resolved without outside intervention.

Case 7: Deadlocks in Disk Sharing

Disks are designed to be shared, so it's not uncommon for two processes to access different areas of the same disk. This ability to share creates an active type of deadlock, known as livelock. Processes use a form of busy-waiting that's different from a natural wait. In this case, it's waiting to share a resource but never actually gains control of it. In Figure 5.6, two competing processes are sending conflicting commands, causing livelock (notice that neither process is blocked, which would cause a deadlock). Instead, each remains active but without achieving any progress and never reaching fulfillment.



(figure 5.6)

Case 7. Two processes are each waiting for an I/O request to be filled: one at track 20 and one at track 310. But by the time the read/write arm reaches one track, a competing command for the other track has been issued, so neither command is satisfied and livelock occurs.

For example, at an insurance company the system performs many daily transactions. One day the following series of events ties up the system:

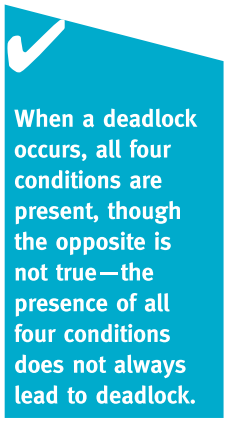
1. A process from Customer Service (P1) wishes to show a payment, so it issues a command to read the balance, which is stored on Track 20 of a disk.
2. While the control unit is moving the arm to Track 20, the Customer Service process is put on hold and the I/O channel is free to process the next I/O request.

3. While the arm is moving into position, Accounts Payable (P2) gains control of the I/O channel and issues a command to write someone else's payment to a record stored on Track 310. If the command is not "locked out," the Accounts Payable process is put on hold while the control unit moves the arm to Track 310.
4. Because the Accounts Payable process is "on hold" while the arm is moving, the channel can be captured again by the customer service process, which reconfirms its command to "read from Track 20."
5. Because the last command from the Accounts Payable process had forced the arm mechanism to Track 310, the disk control unit begins to reposition the arm to Track 20 to satisfy the customer service process. The I/O channel is released because the Customer Service process is once again put on hold, so it can be captured by the accounts payable process, which issues a WRITE command—only to discover that the arm mechanism needs to be repositioned again.

As a result, the arm is in a constant state of motion, moving back and forth between Tracks 20 and 310 as it tries to respond to the two competing commands, but it satisfies neither.

Necessary Conditions for Deadlock or Livelock

In each of these seven cases, the deadlock or livelock involved the interaction of several processes and resources, but each time, it was preceded by the simultaneous occurrence of four conditions that the operating system (or other systems) could have recognized: mutual exclusion, resource holding, no preemption, and circular wait. It's important to remember that each of these four conditions is necessary for the operating system to work smoothly. None of them can be removed easily without causing the system's overall functioning to suffer. Therefore, the system needs to recognize the combination of conditions before they occur.



To illustrate these four conditions, let's revisit the staircase example from the beginning of the chapter to identify the four conditions required for a locked system. (Note that deadlock and livelock share the same requirements, so the following discussion applies to both.)

1. When two people meet on the steps, between landings, they can't pass because the steps can hold only one person at a time. **Mutual exclusion**, the act of allowing only one person (or process) to have access to a step (or a dedicated resource), is the first condition for deadlock.
2. When two people meet on the stairs and each one holds ground and waits for the other to retreat, that is an example of **resource holding** (as opposed to resource sharing), the second condition for deadlock.
3. In this example, each step is dedicated to the climber (or the descender); it is allocated to the holder for as long as needed. This is called **no preemption**, the lack of temporary reallocation of resources, and is the third condition for deadlock.

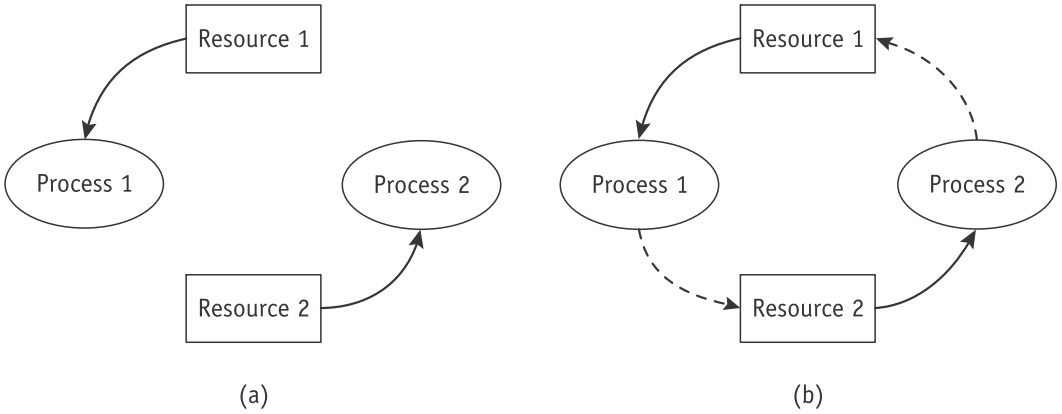
4. These three lead to the fourth condition of **circular wait**, in which each person (or process) involved in the impasse is waiting for another to voluntarily release the step (or resource) so that at least one will be able to continue on and eventually arrive at the destination.

All four conditions are required for the deadlock to occur, and as long as all four conditions are present, the deadlock will continue. However, if one condition is removed, the deadlock will be resolved. In fact, if the four conditions can be prevented from ever occurring at the same time, deadlocks can be prevented. Although this concept may seem obvious, it isn't easy to implement.

Modeling Deadlocks

In 1972, Richard Holt published a visual tool to show how the four conditions can be modeled using **directed graphs**. (We used modified directed graphs in figures shown previously in this chapter.) These graphs use two kinds of symbols: processes represented by circles and resources represented by squares. A solid line with an arrow that runs from a *resource to a process*, as shown in Figure 5.7(a), means that that resource has already been allocated to that process (in other words, the process is holding that resource). A dashed line with an arrow going from a *process to a resource*, as shown in Figure 5.7(b), means that the process is waiting for that resource.

The directions of the arrows are critical because they indicate flow. If for any reason there is an interruption in the flow, then there is no deadlock. On the other hand, if cyclic flow is shown in the graph, (as illustrated in Figure 5.7(b), then there's a deadlock involving the processes and the resources shown in the cycle.



(figure 5.7)

In (a), Resource 1 is being held by Process 1 and Resource 2 is held by Process 2 in a system that is not deadlocked. In (b), Process 1 requests Resource 2 but doesn't release Resource 1, and Process 2 does the same—creating a deadlock. (If one process released its resource, the deadlock would be resolved.)

To practice modeling a system, the next three scenarios evaluate a system with three processes—P1, P2, and P3. The system also has three resources—R1, R2, and R3—each of a different type: printer, disk drive, and plotter. Because there is no specified order in which the requests are handled, we look at three different possible scenarios using graphs to help us detect any deadlocks.

Scenario 1: No Deadlock

The first scenario's sequence of events is shown in Table 5.1. The directed graph is shown in Figure 5.8.

(table 5.1)

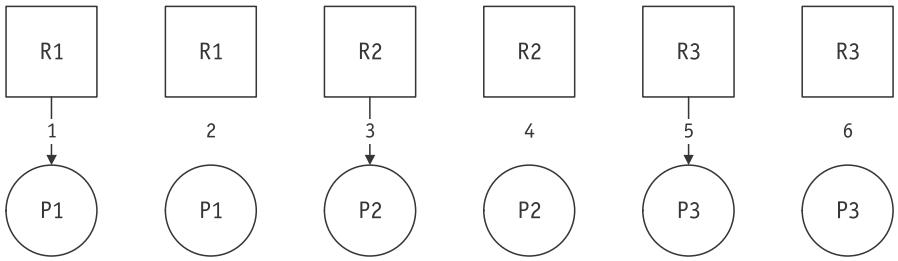
Scenario 1. These events are shown in the directed graph in Figure 5.8.

Event	Action
1	Process 1 (P1) requests and is allocated the printer (R1).
2	Process 1 releases the printer.
3	Process 2 (P2) requests and is allocated the disk drive (R2).
4	Process 2 releases the disk drive.
5	Process 3 (P3) requests and is allocated the plotter (R3).
6	Process 3 releases the plotter.

Notice in the directed graph that no cycles were formed at any time. Therefore, we can safely conclude that a deadlock can't occur with this system at this time, even if each process requests each of the other resources because every resource is released before the next process requests it.

(figure 5.8)

First scenario. The system will stay free of deadlocks if each resource is released before it is requested by the next process.



Scenario 2 – Resource Holding

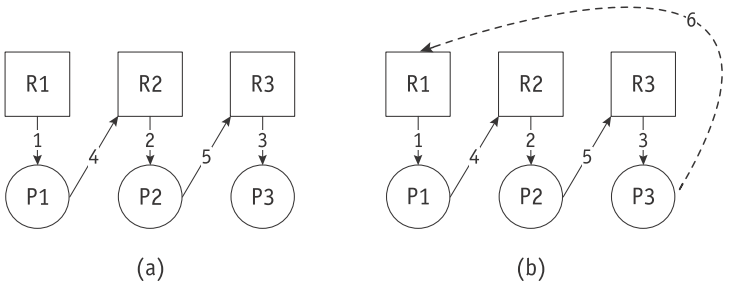
Now, consider a second scenario's sequence of events shown in Table 5.2.

(table 5.2)

Scenario 2. This sequence of events is shown in the two directed graphs shown in Figure 5.9.

Event	Action
1	P1 requests and is allocated R1.
2	P2 requests and is allocated R2.
3	P3 requests and is allocated R3.
4	P1 requests R2.
5	P2 requests R3.
6	P3 requests R1.

The progression of the directed graph is shown in Figure 5.9. A deadlock occurs because every process is waiting for a resource that is being held by another process, but none will be released without intervention.



(figure 5.9)

Second scenario. The system (a) becomes deadlocked (b) when P3 requests R1. Notice the circular wait.

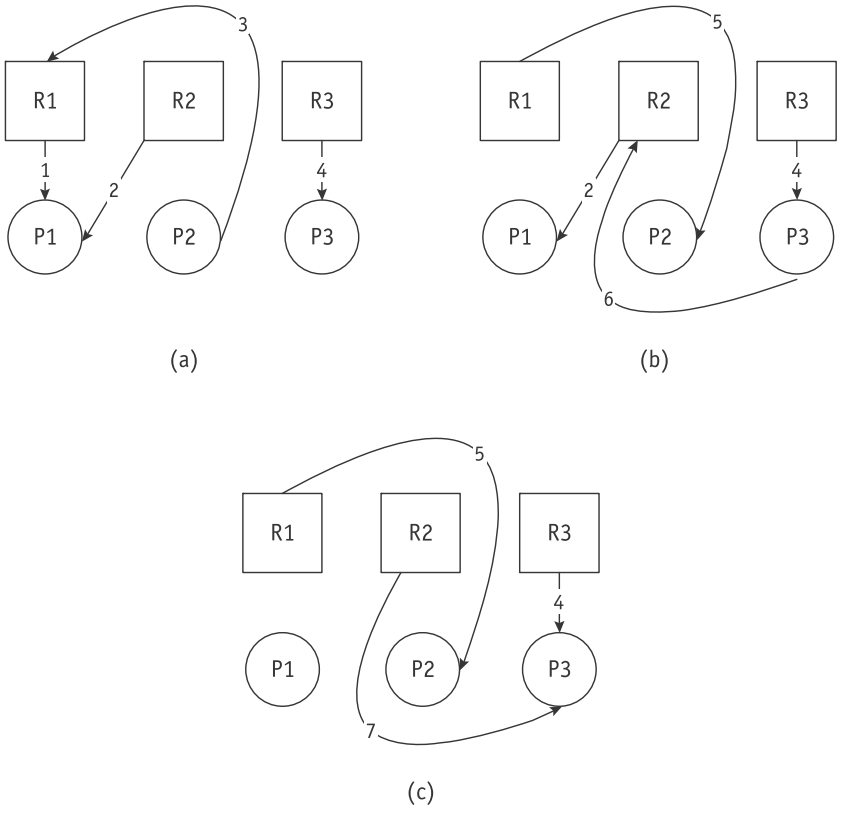
Scenario 3 – Deadlock Broken

The third scenario is shown in Table 5.3. As shown in Figure 5.10, the resources are released before deadlock can occur.

Event	Action
1	P1 requests and is allocated R1.
2	P1 requests and is allocated R2.
3	P2 requests R1.
4	P3 requests and is allocated R3.
5	P1 releases R1, which is allocated to P2.
6	P3 requests R2.
7	P1 releases R2, which is allocated to P3.

(table 5.3)

Scenario 3. This sequence of events is shown in the directed graph in Figure 5.10.



(figure 5.10)

The third scenario. After event 4, the directed graph looks like (a) and P2 is blocked because P1 is holding on to R1. However, event 5 breaks the deadlock and the graph soon looks like (b). Again there is a blocked process, P3, which must wait for the release of R2 in event 7 when the graph looks like (c).

Strategies for Handling Deadlocks

As these examples show, a system's requests and releases can be received in an unpredictable order, which makes it very difficult to design a foolproof preventive policy. In general, operating systems use some combination of several strategies to deal with deadlocks:

- Prevent one of the four conditions from occurring: **prevention**.
- Avoid the deadlock if it becomes probable: **avoidance**.
- Detect the deadlock when it occurs: **detection**.
- Recover from it gracefully: **recovery**.

Prevention

To prevent a deadlock, the operating system must eliminate one of the four necessary conditions discussed at the beginning of this chapter, a task complicated by the fact that the same condition can't be eliminated from every resource.

Mutual exclusion is necessary in any computer system because some resources, such as memory, CPU, and dedicated devices, must be exclusively allocated to one user at a time. In the case of I/O devices, such as printers, the mutual exclusion may be bypassed by spooling, which allows the output from many jobs to be stored in separate temporary spool files at the same time, and each complete output file is then selected for printing when the device is ready. However, we may be trading one type of deadlock (such as Case 3: Deadlocks in Dedicated Device Allocation) for another (Case 5: Deadlocks in Spooling).

Resource holding, where a job holds on to one resource while waiting for another one that's not yet available, could be sidestepped by forcing each job to request, at creation time, every resource it will need to run to completion. For example, if every job in a batch system is given as much memory as it needs, then the number of active jobs will be dictated by how many can fit in memory—a policy that would significantly decrease the degree of multiprogramming. In addition, peripheral devices would be idle because they would be allocated to a job even though they wouldn't be used all the time. As we've said before, this was used successfully in batch environments, although it did reduce the effective use of resources and restricted the amount of multiprogramming. But it doesn't work as well in interactive systems.

No preemption could be bypassed by allowing the operating system to deallocate resources from jobs. This can be done if the state of the job can be easily saved and restored, as when a job is preempted in a round robin environment or a page is swapped to secondary storage in a virtual memory system. On the other hand, preemption of a dedicated I/O device (printer, plotter, scanner, and so on),

or of files during the modification process, can require some extremely unpleasant recovery tasks.

Circular wait can be bypassed if the operating system prevents the formation of a circle. One such solution proposed by Havender (1968) is based on a numbering system for the resources such as: printer = 1, disk = 2, scanner = 3, plotter = 4, and so on. The system forces each job to request its resources in ascending order: any "number one" devices required by the job would be requested first; any "number two" devices would be requested next; and so on. So if a job needed a printer and then a plotter, it would request them in this order: printer (1) first and then the plotter (4). If the job required the plotter first and then the printer, it would still request the printer first (which is a 1) even though it wouldn't be used right away. A job could request a printer (1) and then a disk (2) and then a scanner (3); but if it needed another printer (1) late in its processing, it would still have to anticipate that need when it requested the first one and before it requested the disk.

This scheme of "hierarchical ordering" removes the possibility of a circular wait and therefore guarantees the removal of deadlocks. It doesn't require that jobs state their maximum needs in advance, but it does require that the jobs anticipate the order in

Edsger W. Dijkstra (1930–2002)

Born in Rotterdam, The Netherlands, Edsger Dijkstra worked on many issues that became fundamental to computer science, including those involving recursion, real-time interrupts, the Minimum Spanning Tree Algorithm, and the Dining Philosophers Problem. He codeveloped the first compiler for Algol-60, a high-level programming language of that time. Dijkstra won numerous honors, including the IEEE Computer Society Computer Pioneer Award (1982). He was named a Distinguished Fellow of the British Computer Society (1971) and was one of the first recipients of the ACM Turning Award (1972).

For more information:
http://www.thocp.net/biographies/dijkstra_edsger.htm

Received the ACM 1972 A.M. Turing Award "for fundamental contributions to programming as a high, intellectual challenge; for eloquent insistence and practical demonstration that programs should be composed correctly, not just debugged into correctness; for illuminating perception of problems at the foundations of program design."

(Photo: Hamilton Richards)

which they will request resources. One of the difficulties of this scheme is discovering the best order for the resources so that the needs of the majority of the users are satisfied. Another difficulty is that of assigning a ranking to nonphysical resources such as files or locked database records where there is no basis for assigning a higher number to one over another.

Avoidance

Even if the operating system can't remove one of the conditions for deadlock, it can avoid one if the system knows ahead of time the sequence of requests associated with each of the active processes. As was illustrated in the graphs shown in Figure 5.7 through Figure 5.10, there exists at least one allocation of resources sequence that will allow jobs to continue without becoming deadlocked.

To remain in a safe state, the bank has to have sufficient funds to satisfy the needs of at least one customer.

One such algorithm was proposed by Dijkstra in 1965 to regulate resource allocation to avoid deadlocks. The Banker's Algorithm is based on a bank with a fixed amount of capital that operates on the following principles:

- No customer will be granted a loan exceeding the bank's total capital.
- All customers will be given a maximum credit limit when opening an account.
- No customer will be allowed to borrow over the limit.
- The sum of all loans won't exceed the bank's total capital.

Under these conditions, the bank isn't required to have on hand the total of all maximum lending quotas before it can open up for business (we'll assume the bank will always have the same fixed total and we'll disregard interest charged on loans). For our example, the bank has a total capital fund of $10,000 and has three customers (#1, #2, and #3), who have maximum credit limits of $4,000, $5,000, and $8,000, respectively. Table 5.4 illustrates the state of affairs of the bank after some loans have been granted to customers #2 and #3. This is called a **safe state** because the bank still has enough money left to satisfy the maximum requests of all three customers.

(table 5.4)

The bank started with $10,000 and has remaining capital of $4,000 after these loans. Therefore, it's in a "safe state."

Customer	Loan Amount	Maximum Credit	Remaining Credit
Customer #1	0	4,000	4,000
Customer #2	2,000	5,000	3,000
Customer #3	4,000	8,000	4,000
Total loaned: $6,000			
Total capital fund: $10,000			

A few weeks later after more loans have been made, and some have been repaid, the bank is in the **unsafe state** represented in Table 5.5.

Customer	Loan Amount	Maximum Credit	Remaining Credit
Customer #1	2,000	4,000	2,000
Customer #2	3,000	5,000	2,000
Customer #3	4,000	8,000	4,000
Total loaned: $9,000			
Total capital fund: $10,000			

(table 5.5)

The bank only has remaining capital of $1,000 after these loans and therefore is in an "unsafe state."

This is an unsafe state because with only $1,000 left, the bank can't satisfy anyone's maximum request. Also, if the bank lent the $1,000 to anyone, it would be deadlocked (it wouldn't be able to make the loan). An unsafe state doesn't necessarily lead to deadlock, but it does indicate that the system is an excellent candidate for one. After all, none of the customers is required to request the maximum, but the bank doesn't know the exact amount that will eventually be requested; and as long as the bank's capital is less than the maximum amount available for individual loans, it can't guarantee that it will be able to fill every loan request.

If we substitute jobs for customers and dedicated devices for dollars, we can apply the same principles to an operating system. In the following example, the system has 10 devices. Table 5.6 shows our system in a safe state, and Table 5.7 depicts the same system in an unsafe state. As before, a safe state is one in which there are enough available resources to satisfy the maximum needs of at least one job. Then, using the resources released by the finished job, the maximum needs of another job can be filled and that job can be finished, and so on, until all jobs are done.

Job No.	Devices Allocated	Maximum Required	Remaining Needs
Job 1	0	4	4
Job 2	2	5	3
Job 3	4	8	4
Total number of devices allocated: 6			
Total number of devices in system: 10			

(table 5.6)

Resource assignments after initial allocations. This is a safe state: Six devices are allocated and four units are still available.

The operating system must be sure never to satisfy a request that moves it from a safe state to an unsafe one. Therefore, as users' requests are satisfied, the operating system must identify the job with the smallest number of remaining resources and

(table 5.7)

Resource assignments after later allocations. This is an unsafe state: Only one unit is available but every job requires at least two to complete its execution.

Job No.	Devices Allocated	Maximum Required	Remaining Needs
Job 1	2	4	2
Job 2	3	5	2
Job 3	4	8	4
Total number of devices allocated: 9			
Total number of devices in system: 10			

make sure that the number of available resources is always equal to, or greater than, the number needed for this job to run to completion. Requests that would place the safe state in jeopardy must be blocked by the operating system until they can be safely accommodated.

If the system is always kept in a safe state, all requests are eventually satisfied and a deadlock is avoided.

If this elegant solution is expanded to work with several classes of resources, the system sets up a "resource assignment table" for each type of resource and tracks each table to keep the system in a safe state.

Although the Banker's Algorithm has been used to avoid deadlocks in systems with a few resources, it isn't practical for most systems for several reasons:

- As they enter the system, jobs must predict the maximum number of resources needed. As we've said before, this isn't practical in interactive systems.
- The number of total resources for each class must remain constant. If a device breaks and becomes suddenly unavailable, the algorithm won't work (the system may already be in an unsafe state).
- The number of jobs must remain fixed, something that isn't possible in interactive systems where the number of active jobs is constantly changing.
- The overhead cost incurred by running the avoidance algorithm can be quite high when there are many active jobs and many devices, because the algorithm has to be invoked for every request.
- Resources aren't well utilized because the algorithm assumes the worst case and, as a result, keeps vital resources unavailable to guard against unsafe states.
- Scheduling suffers as a result of the poor utilization and jobs are kept waiting for resource allocation. A steady stream of jobs asking for a few resources can cause the indefinite postponement of a more complex job requiring many resources.

Detection

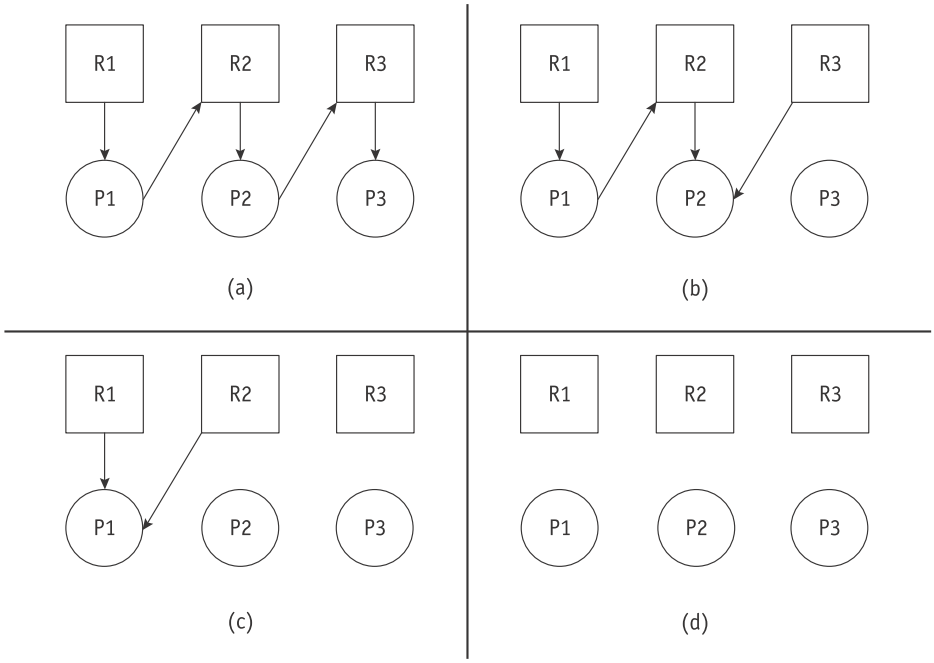
The directed graphs presented earlier in this chapter showed how the existence of a circular wait indicated a deadlock, so it's reasonable to conclude that deadlocks can be detected by building directed resource graphs and looking for cycles. Unlike the

avoidance algorithm, which must be performed every time there is a request, the algorithm used to detect circularity can be executed whenever it is appropriate: every hour, once a day, when the operator notices that throughput has deteriorated, or when an angry user complains.

The detection algorithm can be explained by using directed resource graphs and "reducing" them. Begin with a system that is in use, as shown in Figure 5.11(a). The steps to reduce a graph are these:

1. Find a process that is currently using a resource and *not waiting* for one. This process can be removed from the graph by disconnecting the link tying the resource to the process, such as P3 in Figure 5.11(b). The resource can then be returned to the "available list." This is possible because the process would eventually finish and return the resource.
2. Find a process that's waiting only for resource classes that aren't fully allocated, such as P2 in Figure 5.11(c). This process isn't contributing to deadlock since it would eventually get the resource it's waiting for, finish its work, and return the resource to the "available list" as shown in Figure 5.11(c).
3. Go back to Step 1 and continue with Steps 1 and 2 until all lines connecting resources to processes have been removed, eventually reaching the stage shown in Figure 5.11(d).

If there are any lines left, this indicates that the request of the process in question can't be satisfied and that a deadlock exists. Figure 5.12 illustrates a system in which three processes—P1, P2, and P3—and three resources—R1, R2, and R3—aren't deadlocked.



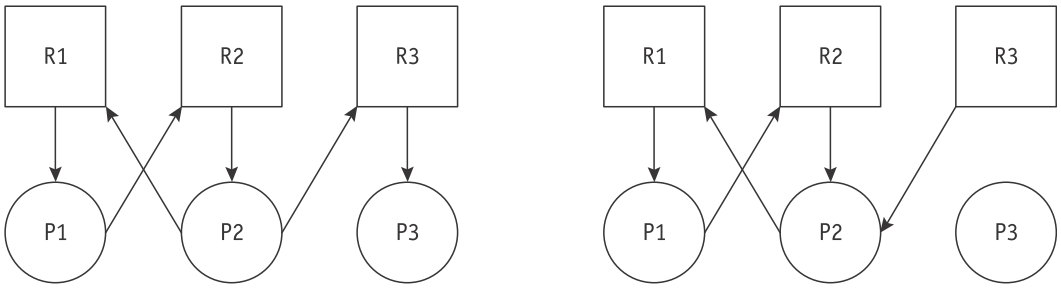
(figure 5.11)

This system is deadlock-free because the graph can be completely reduced, as shown in (d).

Figure 5.11 shows the stages of a graph reduction from (a), the original state. In (b), the link between P3 and R3 can be removed because P3 isn't waiting for any other resources to finish, so R3 is released and allocated to P2 (Step 1). In (c), the links between P2 and R3 and between P2 and R2 can be removed because P2 has all of its requested resources and can run to completion—and then R2 can be allocated to P1. Finally, in (d), the links between P1 and R2 and between P1 and R1 can be removed because P1 has all of its requested resources and can finish successfully. Therefore, the graph is completely resolved.

(figure 5.12)

Even after this graph (a) is reduced as much as possible (by removing the request from P3), it is still deadlocked (b).



The deadlocked system in Figure 5.12 cannot be reduced because of a key element: P2 is linked to R1. In (a), the link between P3 and R3 can be removed because P3 isn't waiting for any other resource, so R3 is released and allocated to P2. But in (b), P2 has only two of the three resources it needs to finish and is waiting for R1. But R1 can't be released by P1, because P1 is waiting for R2, which is held by P2; moreover, P1 can't finish because it is waiting for P2 to finish (and release R2); and P2 can't finish because it's waiting for R1. This is a circular wait.

Recovery

Once a deadlock has been detected, it must be untangled and the system returned to normal as quickly as possible. There are several **recovery** algorithms, but they all have one feature in common: They all require at least one **victim**, an expendable job, which, when removed from the deadlock, will free the system. Unfortunately for the victim, removal generally requires that the job be restarted from the beginning or from a convenient midpoint.

The first and simplest recovery method, and the most drastic, is to terminate every job that's active in the system and restart them from the beginning.

The second method is to terminate only the jobs involved in the deadlock and ask their users to resubmit them.

The third method is to identify which jobs are involved in the deadlock and terminate them one at a time, checking to see if the deadlock is eliminated after each removal, until the deadlock has been resolved. Once the system is freed, the remaining jobs are allowed to complete their processing, and later, the halted jobs are started again from the beginning.

The fourth method can be put into effect only if the job keeps a record, a snapshot, of its progress so it can be interrupted and then continued without starting again from the beginning of its execution. The snapshot is like the landing in our staircase example: Instead of forcing the deadlocked stair climbers to return to the bottom of the stairs, they need to retreat only to the nearest landing and wait until the others have passed. Then the climb can be resumed. In general, this method is favored for long-running jobs to help them make a speedy recovery.

Until now we've offered solutions involving the jobs caught in the deadlock. The next two methods concentrate on the nondeadlocked jobs and the resources they hold. One of them, the fifth method in our list, selects a nondeadlocked job, preempts the resources it's holding, and allocates them to a deadlocked process so it can resume execution, thus breaking the deadlock. The sixth method stops new jobs from entering the system, which allows the nondeadlocked jobs to run to completion so they'll release their resources. Eventually, with fewer jobs in the system, competition for resources is curtailed so the deadlocked processes get the resources they need to run to completion. This method is the only one listed here that doesn't rely on a victim, and it's not guaranteed to work unless the number of available resources surpasses that needed by at least one of the deadlocked jobs to run (this is possible with multiple resources).

Several factors must be considered to select the victim that will have the least-negative effect on the system. The most common are:

- The priority of the job under consideration—high-priority jobs are usually untouched.
- CPU time used by the job—jobs close to completion are usually left alone.
- The number of other jobs that would be affected if this job were selected as the victim.

In addition, programs working with databases also deserve special treatment because a database that is only partially updated is only partially correct. Therefore, jobs that are modifying data shouldn't be selected for termination because the consistency and validity of the database would be jeopardized. Fortunately, designers of many database systems have included sophisticated recovery mechanisms so damage to the database is minimized if a transaction is interrupted or terminated before completion.

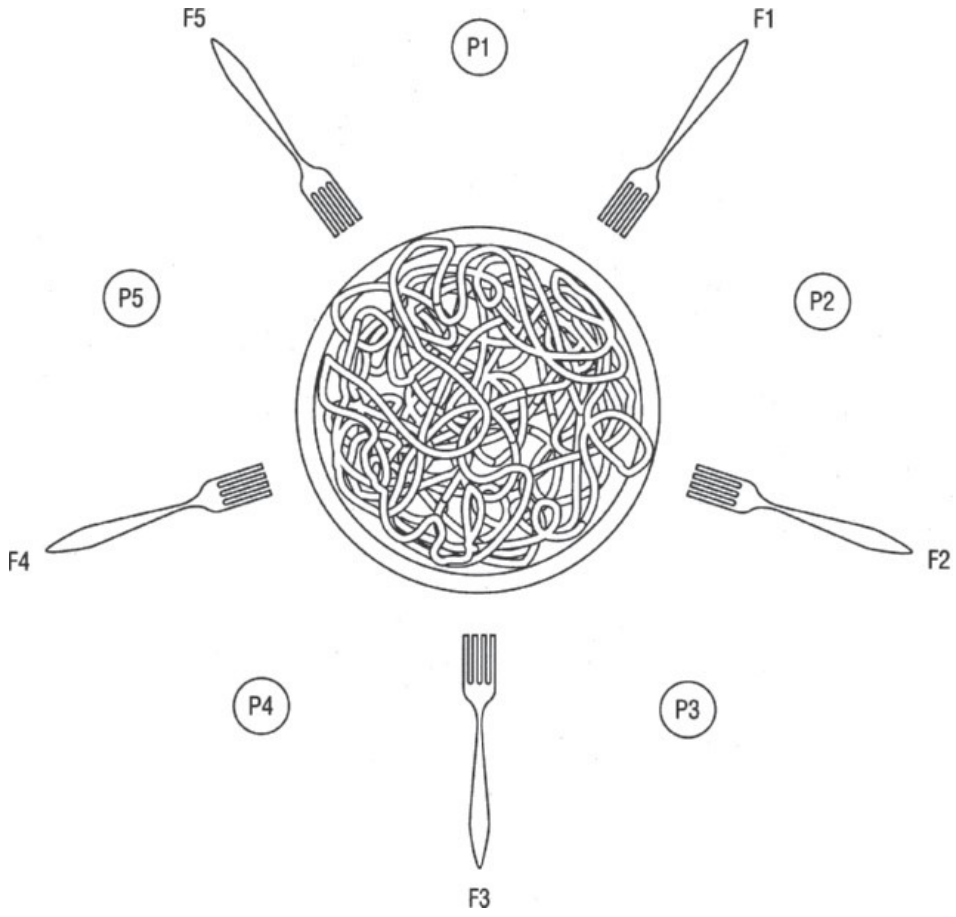
Starvation

While deadlock affects system-wide performance, starvation affects individual jobs or processes. To find the starved tasks, the system monitors the waiting times for PCBs in the WAITING queues.

So far in this chapter, we have concentrated on deadlocks and livelocks, both of which result from the liberal allocation of resources. At the opposite end is starvation, the result of conservative allocation of resources, where a single job is prevented from execution because it's kept waiting for resources that never become available. To illustrate this, the case of the Dining Philosophers Problem was introduced by Dijkstra in 1968.

Five philosophers are sitting at a round table, each deep in thought. In the center of the table, accessible to everyone, is a bowl of spaghetti. There are five forks on the table—one between each philosopher, as illustrated in Figure 5.13. Local custom dictates that each philosopher must use two forks, the forks on either side of the plate, to eat the spaghetti, but there are only five forks—not the 10 it would require for all five thinkers to eat at once—and that's unfortunate for Philosopher 2.

When they sit down to dinner, Philosopher 1 (P1) is the first to take the two forks (both F1 and F5) on either side of the plate and begins to eat. Inspired by this bold move, Philosopher 3 (P3) does likewise, acquiring both F2 and F3. Now Philosopher 2 (P2)



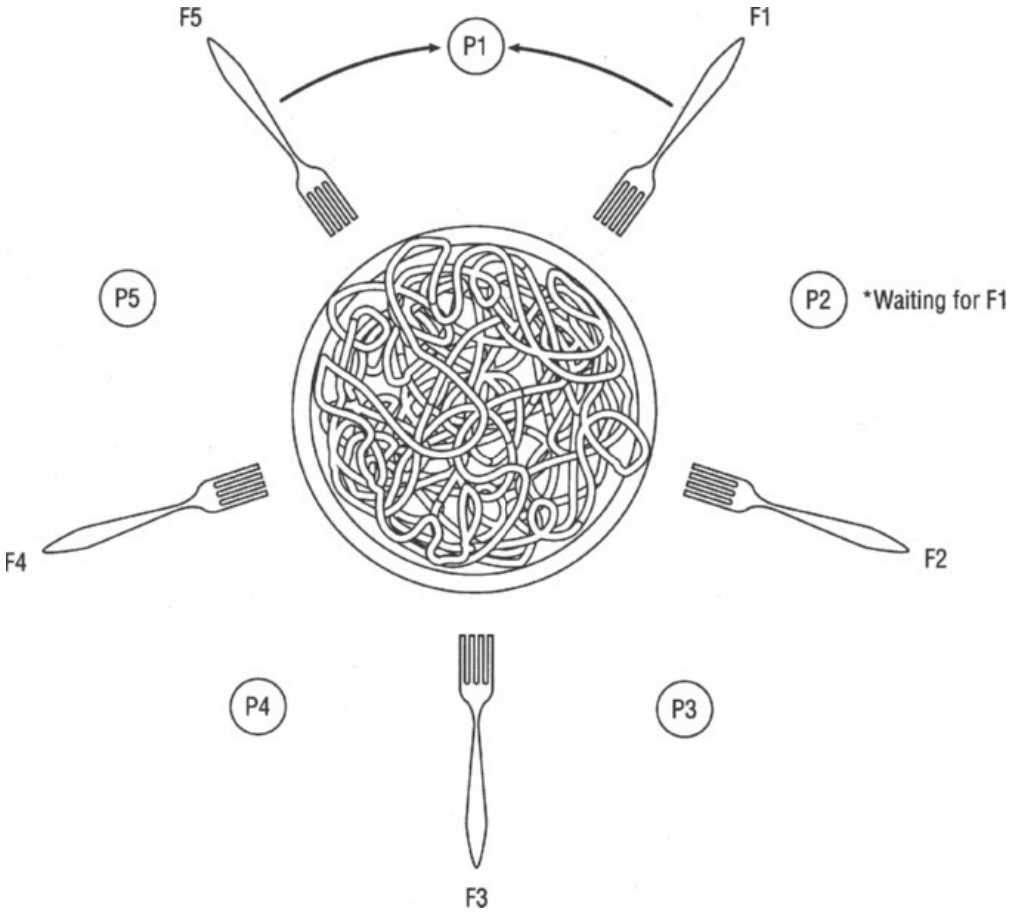
(figure 5.13)

The dining philosophers' table, before the meal begins.

decides to begin the meal, but is unable to begin because no forks are available: Fork #1 has been allocated to Philosopher 1, and Fork #2 has been allocated to Philosopher 3. In fact, by this time, four of the five forks have been allocated. The only remaining fork is situated between Philosopher 4 and Philosopher 5. So Philosopher 2 must wait.

Soon, Philosopher 3 finishes eating, puts down the two forks, and resumes pondering. Should the fork that's now free, Fork #2, be allocated to the hungry Philosopher 2, even though only that one fork is free? Although it's tempting, such a move would be a bad precedent, because if the philosophers are allowed to tie up resources with only the hope that the other required resource will become available, the dinner could easily slip into an unsafe state; it would be only a matter of time before each philosopher held a single fork, and nobody could eat. So the resources are allocated to the philosophers only when both forks are available at the same time. The status of the "system" is illustrated in Figure 5.14.

P4 and P5 are quietly thinking and Philosopher 1 is still eating. Philosopher 3 (who should be full) decides to eat some more and is able to take F2 and F3 once again. Soon thereafter, Philosopher 1 finishes and releases Fork #1 and Fork #5, but Philosopher 2



(figure 5.14)

Each philosopher must have both forks to begin eating, the one on the right and the one on the left. Unless the resources, the forks, are allocated fairly, some philosophers may starve.

is still not able to eat, because Fork #2is now allocated. This scenario could continue forever; as long as Philosopher 1 and Philosopher 3 alternate their use of the available resources, Philosopher 2 must wait. Philosophers 1 and 3 can eat any time they wish, while Philosopher 2 starves, only inches from nourishment.

In a computer environment, the resources are like forks and the competing processes are like dining philosophers. If the resource manager doesn't watch for starving processes and jobs and plan for their eventual completion, they could remain in the system forever waiting for the right combination of resources.

To address this problem, an algorithm designed to detect starving jobs can be implemented, which tracks how long each job has been waiting for resources (this is the same as aging, described in Chapter 4). Once starvation has been detected, the system can block new jobs until the starving jobs have been satisfied. This algorithm must be monitored closely: If monitoring is done too often, then new jobs will be blocked too frequently and throughput will be diminished. If it's not done often enough, then starving jobs will remain in the system for an unacceptably long period of time.

Conclusion

Every operating system must dynamically allocate a limited number of resources while avoiding the two extremes of deadlock and starvation.

In this chapter we discussed several methods of dealing with both livelock and deadlock: prevention, avoidance, detection, and recovery. Deadlocks can be prevented by not allowing the four conditions of a deadlock to occur in the system at the same time. By eliminating at least one of the four conditions (mutual exclusion, resource holding, no preemption, and circular wait), the system can be kept deadlock-free. As we've seen, the disadvantage of a preventive policy is that each of these conditions is vital to different parts of the system at least some of the time, so prevention algorithms are complex, and to routinely execute them requires significant overhead.

Deadlocks can be avoided by clearly identifying safe states and unsafe states and requiring the system to keep enough resources in reserve to guarantee that all jobs active in the system can run to completion. The disadvantage of an avoidance policy is that the system's resources aren't allocated to their fullest potential.

If a system doesn't support prevention or avoidance, then it must be prepared to detect and recover from the deadlocks that occur. Unfortunately, this option usually relies on the selection of at least one "victim"—a job that must be terminated before it finishes execution and then be restarted from the beginning.

In the next chapter, we look at problems related to the synchronization of processes in a multiprocessing environment.

Key Terms

avoidance: the dynamic strategy of deadlock avoidance that attempts to ensure that resources are never allocated in such a way as to place a system in an unsafe state.

circular wait: one of four conditions for deadlock through which each process involved is waiting for a resource being held by another; each process is blocked and can't continue, resulting in deadlock.

deadlock: a problem occurring when the resources needed by some jobs to finish execution are held by other jobs, which, in turn, are waiting for other resources to become available.

detection: the process of examining the state of an operating system to determine whether a deadlock exists.

directed graphs: a graphic model representing various states of resource allocations.

livelock: a locked system whereby two or more processes continually block the forward progress of the others without making any forward progress themselves. It is similar to a deadlock except that neither process is blocked or obviously waiting; both are in a continuous state of change.

locking: a technique used to guarantee the integrity of the data in a database through which the user locks out all other users while working with the database.

mutual exclusion: one of four conditions for deadlock in which only one process is allowed to have access to a resource.

no preemption: one of four conditions for deadlock in which a process is allowed to hold on to resources while it is waiting for other resources to finish execution.

prevention: a design strategy for an operating system where resources are managed in such a way that some of the necessary conditions for deadlock do not hold.

process synchronization: (1) the need for algorithms to resolve conflicts between processors in a multiprocessing environment; or (2) the need to ensure that events occur in the proper order even if they are carried out by several processes.

race: a synchronization problem between two processes vying for the same resource.

recovery: the steps that must be taken, when deadlock is detected, by breaking the circle of waiting processes.

resource holding: one of four conditions for deadlock in which each process refuses to relinquish the resources it holds until its execution is completed even though it isn't using them because it's waiting for other resources.

safe state: the situation in which the system has enough available resources to guarantee the completion of at least one job running on the system.

spooling: a technique developed to speed I/O by collecting in a disk file either input received from slow input devices or output going to slow output devices, such as printers.

starvation: the result of conservative allocation of resources in which a single job is prevented from execution because it's kept waiting for resources that never become available.

unsafe state: a situation in which the system has too few available resources to guarantee the completion of at least one job running on the system. It can lead to deadlock.

victim: an expendable job that is selected for removal from a deadlocked system to provide more resources to the waiting jobs and resolve the deadlock.

Interesting Searches

For more background on a few of the topics discussed in this chapter, begin a search with these terms:

- False Deadlock Detection
- Starvation and Livelock Detection
- Distributed Deadlock Detection
- Deadlock Resolution Algorithms
- Operating System Freeze

Exercises

Research Topics

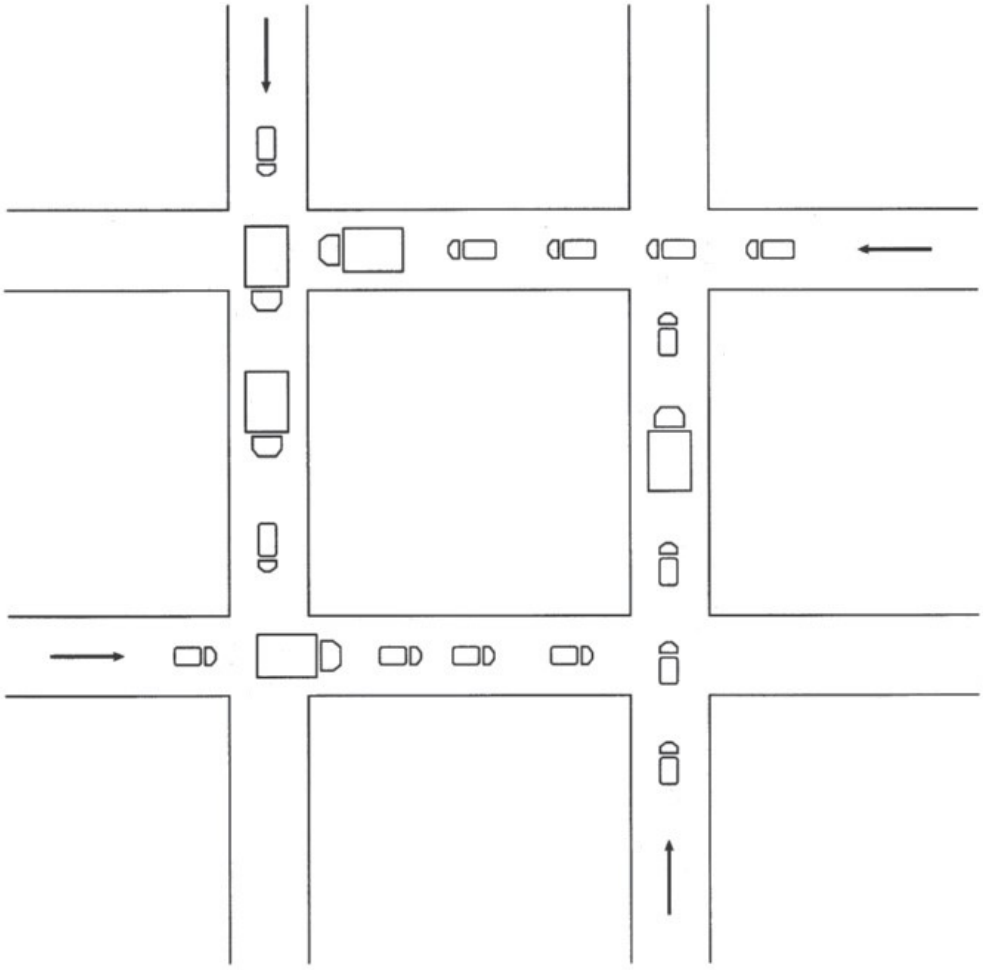
A. In Chapter 3 we introduced the problem of thrashing. Research current literature to investigate the role of deadlock and any resulting thrashing. Discuss how you would begin to quantify the cost to the system (in terms of throughput and performance) of deadlock-caused thrashing. Cite your sources.

B. Research the problem of livelock in a networked environment. Describe how its consequences differ from those of deadlock, and give a real-life example of the problem that's not mentioned in this chapter. Identify at least two different methods the operating system could use to detect and resolve livelock. Cite your sources.

Exercises

1. For each of these conditions— deadlock, race, and starvation—give at least two "real life" examples (not related to a computer system environment) of each of these concepts. Then give your own opinion on how each of these six conditions can be resolved.
2. Given the ice cream sundae example from the beginning of this chapter, identify which elements of the deadlock represent the four necessary conditions for this deadlock.

3. Using the narrow staircase example from the beginning of this chapter, create a list of actions or tasks that could be implemented by the building manager that would allow people to use the staircase without risking a deadlock or starvation.
4. If a deadlock occurs at a combination of downtown intersections, as shown in the figure below, explain in detail how you would identify that a deadlock has occurred, how you would resolve it after it happens, and how you would act to prevent it from happening again.

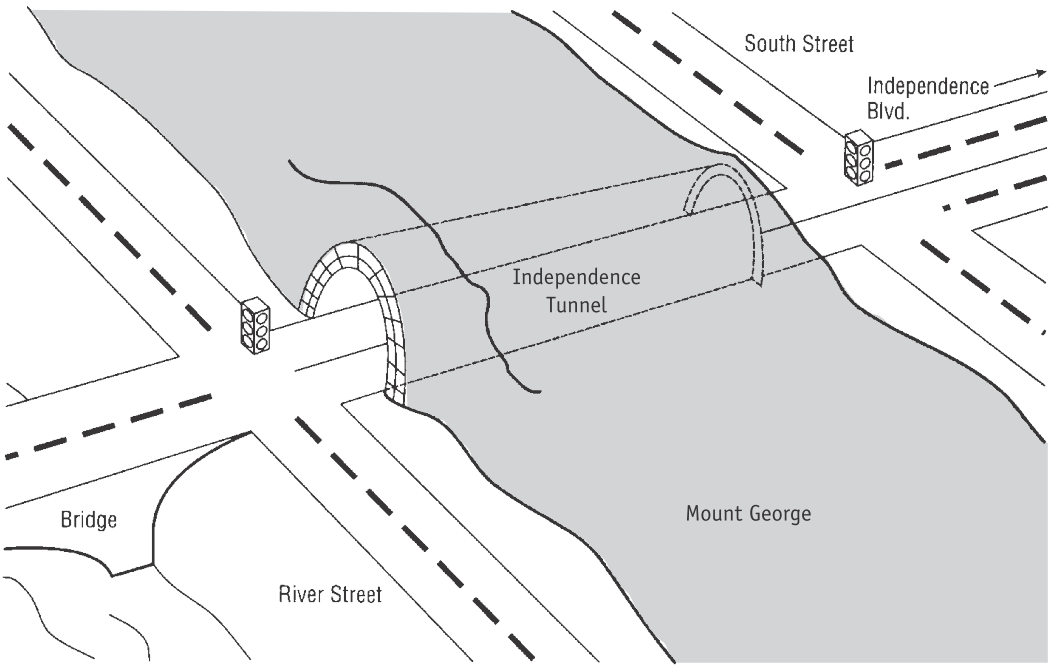


A classic case of traffic deadlock on four one-way streets. This is "gridlock," where no vehicles can move forward to clear the traffic jam.

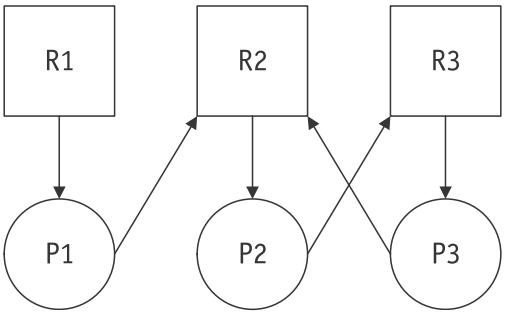
5. Regarding the role played by the "victim" in deadlock resolution, explain your answers to these questions.
   a. How is the victim chosen?
   b. What is the fate of the victim?
   c. Describe the actions required, if any, to complete the victim's tasks.

6. The figure below shows a tunnel going through a mountain and two streets parallel to each other—one at each end of the tunnel. Traffic lights are located at each end of the tunnel to control the cross flow of traffic through each intersection. Based on this figure, answer the following questions:
   a. How can deadlock occur, and under what circumstances?
   b. How can deadlock be detected?
   c. Give a solution to prevent deadlock and starvation.

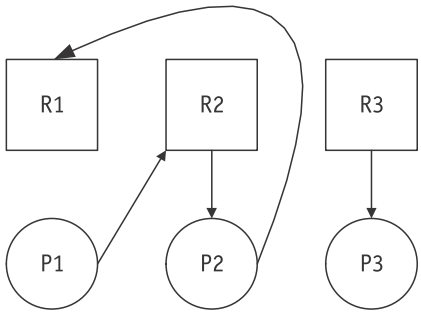
Traffic flow diagram for Exercise 6.



7. Consider the directed graph shown below and answer the following questions:
   a. Is this system deadlocked?
   b. Which, if any, processes are blocked?
   c. What is the resulting graph after reduction?



8. Consider the directed resource graph shown below and answer the following questions:
   a. Is this system deadlocked?
   b. Which, if any, processes are blocked?
   c. What is the resulting graph after reduction?



9. Consider a system with 13 dedicated devices of the same type. All jobs currently running on this system require a maximum of three devices to complete their execution. The jobs run for long periods of time with just two devices, requesting the remaining device only at the very end of the run. Assume that the job stream is endless and that your operating system's device allocation policy is a very conservative one: No job will be started unless all the required drives have been allocated to it for the entire duration of its run.
   a. What is the maximum number of jobs that can be in progress at once? Explain your answer.
   b. What are the minimum and maximum number of devices that may be idle as a result of this policy? Under what circumstances would an additional job be started?
10. Consider a system with 14 dedicated devices of the same type. All jobs currently running on this system require a maximum of five devices to complete their execution. The jobs run for long periods of time with just three devices, requesting the remaining two only at the very end of the run. Assume that the job stream is endless and that your operating system's device allocation policy is a very conservative one: No job will be started unless all the required drives have been allocated to it for the entire duration of its run.
   a. What is the maximum number of jobs that can be active at once? Explain your answer.
   b. What are the minimum and maximum number of devices that may be idle as a result of this policy? Under what circumstances would an additional job be started?

11. Suppose your system is identical to the one described in the previous exercise with 14 devices, but supports the Banker's Algorithm.
    a. What is the maximum number of jobs that can be in progress at once? Explain your answer.
    b. What are the minimum and maximum number of devices that may be idle as a result of this policy? Under what circumstances would an additional job be started?
12. Suppose your system is identical to the one described in the previous exercise, but has 16 devices and supports the Banker's Algorithm.
    a. What is the maximum number of jobs that can be in progress at once? Explain your answer.
    b. What are the minimum and maximum number of devices that may be idle as a result of this policy? Under what circumstances would an additional job be started?

13-16. For the systems described in Questions 13 through 16 below, given that all of the devices are of the same type, and using the definitions presented in the discussion of the Banker's Algorithm, answer these questions:
    a. Calculate the number of available devices.
    b. Determine the remaining needs for each job in each system.
    c. Determine whether each system is safe or unsafe.
    d. If the system is in a safe state, list the sequence of requests and releases that will make it possible for all processes to run to completion.
    e. If the system is in an unsafe state, show how it's possible for deadlock to occur.

13. This system has 16 devices.

Job No.	Devices Allocated	Maximum Required	Remaining Needs
Job 1	5	8	
Job 2	3	9	
Job 3	4	8	
Job 4	2	5	

14. This system has 12 devices.

Job No.	Devices Allocated	Maximum Required	Remaining Needs
Job 1	5	8	
Job 2	1	4	
Job 3	5	7	

15. This system has 14 devices.

Job No.	Devices Allocated	Maximum Required	Remaining Needs
Job 1	2	6	
Job 2	4	7	
Job 3	5	6	
Job 4	0	2	
Job 5	2	4	

16. This system has 32 devices.

Job No.	Devices Allocated	Maximum Required	Remaining Needs
Job 1	11	17	
Job 2	7	10	
Job 3	12	18	
Job 4	0	8	

Advanced Exercises

17. Explain how you would design and implement a mechanism to allow the operating system to detect which, if any, processes are starving.
18. As mentioned in this chapter, Havender proposed a hierarchical ordering of resources, such as disks, printers, files, ports, and so on. Explain in detail the limitations imposed on programs (and on systems) that have to follow such a hierarchical ordering system. Think of some circumstance for which would you recommend a hierarchical ordering system, and explain your answer.
17. Suppose you are an operating system designer and have been approached by the system administrator to help solve the recurring deadlock problem in your installation's spooling system. What features might you incorporate into the operating system so that deadlocks in the spooling system can be resolved without losing the work (the system processing) that was underway or already performed by the deadlocked processes?
19. Given the four primary types of resources—CPU, main memory, storage devices, and files—select for each one the most suitable technique described in this chapter to fight deadlock, and explain why that is your choice.

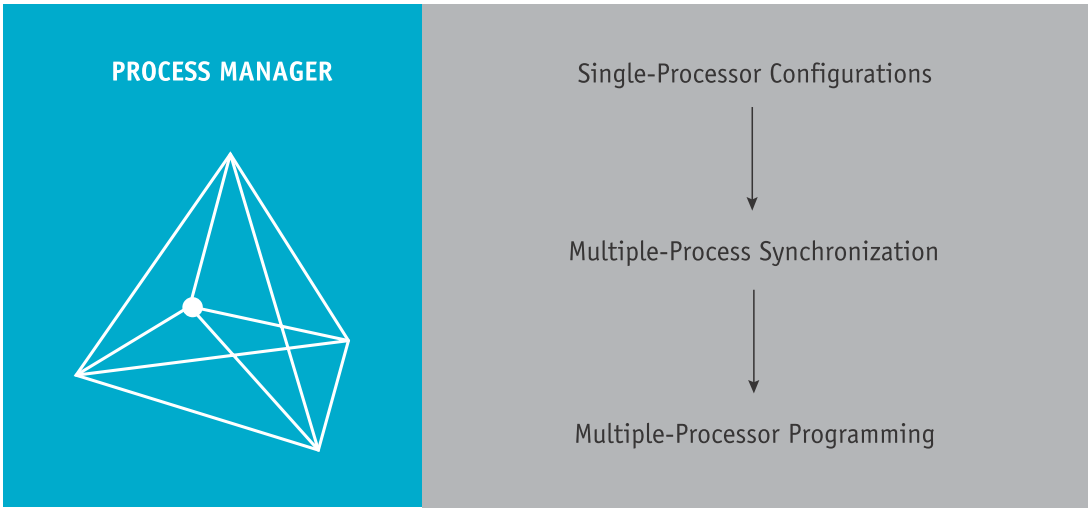
20. Consider a banking system with 10 accounts. Funds may be transferred between two of those accounts by following these steps:

lock A(i);	lock A(j);
update A(i);	update A(j);
unlock A(i);	unlock A(j);

a. Can this system become deadlocked? If yes, show how. If no, explain why not.

b. Could the numbering request policy (presented in the chapter discussion about detection) be implemented to prevent deadlock if the number of accounts is dynamic? Explain why or why not.

Chapter 6 Concurrent Processes



"The measure of power is obstacles overcome."

—Oliver Wendell Holmes, Jr. (1841–1935)

Learning Objectives

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

- The critical difference between processes and processors and how they're connected
- The differences among common configurations of multiprocessing systems
- The basic concepts of multicore processor technology
- The significance of a critical region in process synchronization
- The essential ideas behind process synchronization software
- The need for process cooperation when several processors work together
- How processors cooperate when executing a job, process, or thread
- The significance of concurrent programming languages and their applications

In Chapters 4 and 5, we describe multiprogramming systems that use only one CPU, one processor, which is shared by several jobs or processes. This is called *multiprogramming*. In this chapter we look at another common situation, **multiprocessing** systems, which have several processors working together.

What Is Parallel Processing?

Parallel processing is a situation in which two or more processors operate in one system at the same time and may or may not work on related activities. In other words, two or more CPUs execute instructions simultaneously and the Processor Manager has to coordinate activities of each processor while synchronizing the cooperative interactions among all of them.

There are two primary benefits to parallel processing systems: increased reliability and faster processing.

The reliability stems from the availability of more than one CPU. Theoretically, if one processor fails, then the others can continue to operate and absorb the load. However, this capability must be designed into the system so that, first, the failing processor informs other processors to take over, and second, the operating system dynamically restructures its available resources and allocation strategies so that the remaining processors don't become overloaded.

The increased processing speed can be achieved when instructions or data manipulation can be processed in parallel, two or more at a time. Some systems allocate a CPU to each program or job. Others allocate a CPU to each working set or parts of it. Still others subdivide individual instructions so that each subdivision can be processed simultaneously (called concurrent programming). And others achieve parallel processing by allocating several CPUs to perform a set of instructions separately on vast amounts of data and combine all the results at the end of the job.

Increased flexibility brings increased complexity, however, and two major challenges remain: how to connect the processors into workable configurations and how to orchestrate their interaction to achieve efficiencies, which applies to multiple interacting processes as well. (It might help if you think of each process as being run on a separate processor.)

The complexities of the Processor Manager's multiprocessing tasks are easily illustrated with an example: You're late for an early afternoon appointment and you're in danger of missing lunch, so you get in line for the drive-through window of the local fast-food shop. When you place your order, the order clerk confirms your request, tells you how much it will cost, and asks you to drive to the pickup window; there, a cashier collects your money and hands over your order. All's well and once again

you're on your way—driving and thriving. You just witnessed a well-synchronized multiprocessing system. Although you came in contact with just two processors (the order clerk and the cashier), there were at least two other processors behind the scenes who cooperated to make the system work—the cook and the bagger.

A fast food lunch spot is similar to the six-step information retrieval system below. It is described in a different way in Table 6.1.

a) Processor 1 (the order clerk) accepts the query, checks for errors, and passes the request on to Processor 2 (the bagger).

b) Processor 2 (the bagger) searches the database for the required information (the hamburger).

c) Processor 3 (the cook) retrieves the data from the database (the meat to cook for the hamburger) if it's kept off-line in secondary storage.

d) Once the data is gathered (the hamburger is cooked), it's placed where Processor 2 can get it (in the hamburger bin).

e) Processor 2 (the bagger) retrieves the data (the hamburger) and passes it on to Processor 4 (the cashier).

f) Processor 4 (the cashier) routes the response (your order) back to the originator of the request—you.

(table 6.1)

The six steps of the four-processor fast food lunch stop.

Originator	Action	Receiver
Processor 1 (the order clerk)	Accepts the query, checks for errors, and passes the request on to the receiver	Processor 2 (the bagger)
Processor 2 (the bagger)	Searches the database for the required information (the hamburger)	
Processor 3 (the cook)	Retrieves the data from the database (the meat to cook for the hamburger) if it's kept off-line in secondary storage	
Processor 3 (the cook)	Once the data is gathered (the hamburger is cooked), it's placed where the receiver can get it (in the hamburger bin)	Processor 2 (the bagger)
Processor 2 (the bagger)	Retrieves the data (the hamburger) and passes it on to the receiver	Processor 4 (the cashier)
Processor 4 (the cashier)	Routes the response (your order) back to the originator of the request	You

Synchronization is the key to any multiprocessing system's success because many things can go wrong. For example, what if the microphone broke down and you couldn't speak with the order clerk? What if the cook produced hamburgers at full speed all day, even during slow periods? What would happen to the extra hamburgers? What if the cook became badly burned and couldn't cook anymore? What would the

bagger do if there were no hamburgers? What if the cashier took your money but didn't give you any food? Obviously, the system can't work properly unless every processor communicates and cooperates with every other processor.

Levels of Multiprocessing

Multiprocessing can take place at several different levels, each of which tends to require a different frequency of synchronization, as shown in Table 6.2. At the job level, multiprocessing is fairly benign. It's as if each job is running on its own workstation with shared system resources. On the other hand, when multiprocessing takes place at the thread level, a high degree of synchronization is required to disassemble each process, perform the thread's instructions, and then correctly reassemble the process. This may require additional work by the programmer, as we see later in this chapter.

Parallelism Level	Process Assignments	Synchronization Required
Job Level	Each job has its own processor, and all processes and threads are run by that same processor.	No explicit synchronization required after jobs are assigned to a processor.
Process Level	Unrelated processes, regardless of job, can be assigned to available processors.	Moderate amount of synchronization required.
Thread Level	Threads, regardless of job and process, can be assigned to available processors.	High degree of synchronization required to track each process and each thread.

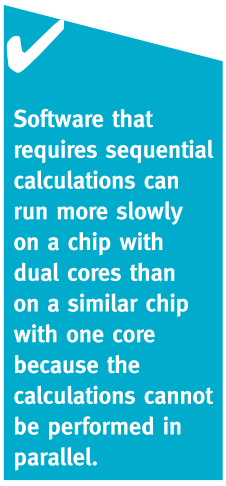
(table 6.2)

Typical levels of parallelism and the required synchronization among processors

Introduction to Multi-Core Processors

Multi-core processors have several processors on a single chip. As processors became smaller in size (as predicted by Moore's Law) and faster in processing speed, CPU designers began to use nanometer-sized transistors (one nanometer is one billionth of a meter). Each transistor switches between two positions—0 and 1—as the computer conducts its binary arithmetic at increasingly fast speeds. However, as transistors reached nano-sized dimensions and the space between transistors became ever closer, the quantum physics of electrons got in the way.

In a nutshell, here's the problem. When transistors are placed extremely close together, electrons can spontaneously tunnel, at random, from one transistor to another, causing a tiny but measurable amount of current to leak. The smaller the transistor, the more significant the leak. (When an electron does this "tunneling," it seems to spontaneously disappear from one transistor and appear in another nearby transistor. It's as if a Star Trek voyager asked the electron to be "beamed aboard" the second transistor.)



Software that requires sequential calculations can run more slowly on a chip with dual cores than on a similar chip with one core because the calculations cannot be performed in parallel.

A second problem was the heat generated by the chip. As processors became faster, the heat also climbed and became increasingly difficult to disperse. These heat and tunneling issues threatened to limit the ability of chip designers to make processors ever smaller.

One solution was to create a single chip (one piece of silicon) with two "processor cores" in the same amount of space. With this arrangement, two sets of calculations can take place at the same time. The two cores on the chip generate less heat than a single core of the same size, and tunneling is reduced; however, the two cores each run more slowly than the single core chip. Therefore, to get improved performance from a dual-core chip, the software has to be structured to take advantage of the double calculation capability of the new chip design. Building on their success with two-core chips, designers have created multi-core processors with more than 80 cores on a single chip. An example is shown in Chapter 1.

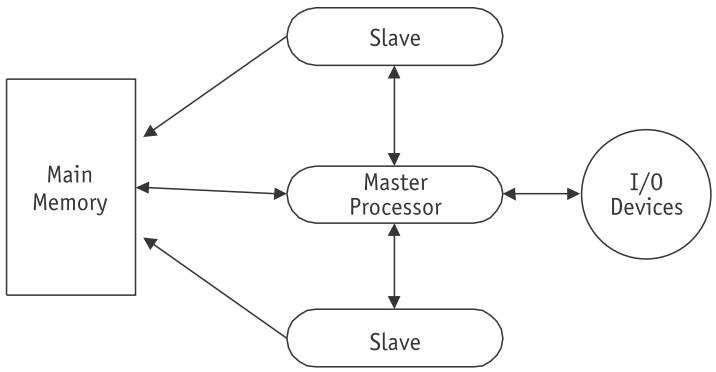
Does this hardware innovation affect the operating system? Yes, because it must manage multiple processors, multiple units of cache and RAM, and the processing of many tasks at once. However, a dual-core chip is not always faster than a single-core chip. It depends on the tasks being performed and whether they're multi-threaded or sequential.

Typical Multiprocessing Configurations

Much depends on how the multiple processors are configured within the system. Three typical configurations are master/slave, loosely coupled, and symmetric.

Master/Slave Configuration

The **master/slave** configuration is an asymmetric multiprocessing system. Think of it as a single-processor system with additional slave processors, each of which is managed by the primary master processor as shown in Figure 6.1.



(figure 6.1)

In a master/slave multiprocessing configuration, slave processors can access main memory directly but they must send all I/O requests through the master processor.

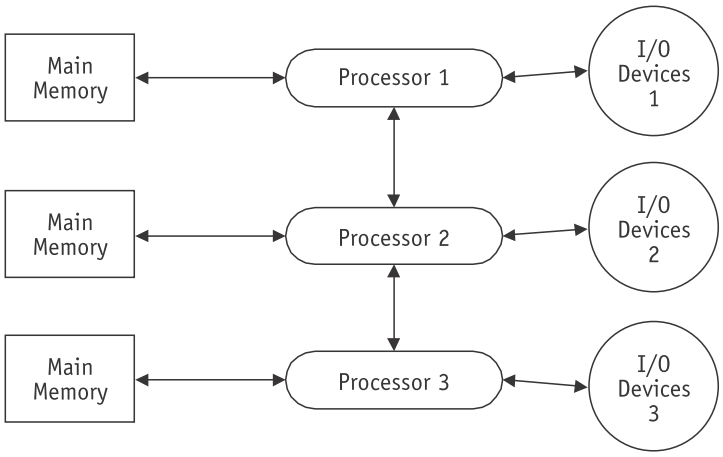
The master processor is responsible for managing the entire system—all files, devices, memory, and processors. Therefore, it maintains the status of all processes in the system, performs storage management activities, schedules the work for the other processors, and executes all control programs. This configuration is well suited for computing environments in which processing time is divided between front-end and back-end processors; in these cases, the front-end processor takes care of the interactive users and quick jobs, and the back-end processor takes care of those with long jobs using the batch mode.

The primary advantage of this configuration is its simplicity. However, it has three serious disadvantages:

- Its reliability is no higher than for a single-processor system because if the master processor fails, none of the slave processors can take over, and the entire system fails.
- It can lead to poor use of resources because if a slave processor should become free while the master processor is busy, the slave must wait until the master becomes free and can assign more work to it.
- It increases the number of interrupts because all slave processors must interrupt the master processor every time they need operating system intervention, such as for I/O requests. This creates long queues at the master processor level when there are many processors and many interrupts.

Loosely Coupled Configuration

The **loosely coupled configuration** features several complete computer systems, each with its own memory, I/O devices, CPU, and operating system, as shown in Figure 6.2. This configuration is called loosely coupled because each processor controls its own resources—its own files, access to memory, and its own I/O devices—and that means that each processor maintains its own commands and I/O management tables.



(figure 6.2)

In a loosely coupled multiprocessing configuration, each processor has its own dedicated resources.

The only difference between a loosely coupled multiprocessing system and a collection of independent single-processing systems is that each processor can communicate and cooperate with the others.

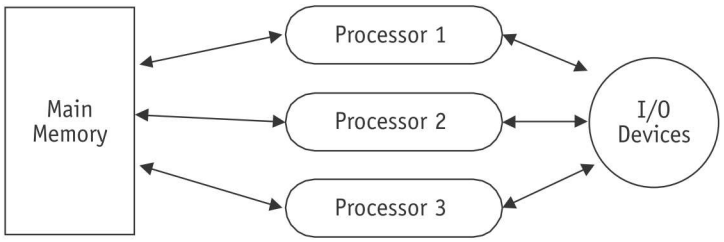
When a job arrives for the first time, it's assigned to one processor. Once allocated, the job remains with the same processor until it's finished. Therefore, each processor must have global tables that indicate where each job has been allocated.

To keep the system well balanced and to ensure the best use of resources, job scheduling is based on several requirements and policies. For example, new jobs might be assigned to the processor with the lightest load or the one with the best combination of output devices available.

This system isn't prone to catastrophic system failures because even when a single processor fails, the others can continue to work independently. However, it can be difficult to detect when a processor has failed.

Symmetric Configuration

A **symmetric configuration** (as depicted in Figure 6.3) features decentralized processor scheduling. That is, a single copy of the operating system and a global table listing each process and its status is stored in a common area of memory so every processor has access to it. Each processor uses the same scheduling algorithm to select which process it will run next.



(figure 6.3)
A symmetric multiprocessing configuration with homogeneous processors. Processes must be carefully synchronized to avoid deadlocks and starvation.

The symmetric configuration (sometimes called tightly coupled) has four advantages over loosely coupled configuration:

- It's more reliable.
- It uses resources effectively.
- It can balance loads well.
- It can degrade gracefully in the event of a failure.

The symmetric configuration is best implemented if all of the processors are of the same type.

However, it is the most difficult configuration to implement because the processes must be well synchronized to avoid the problems of races and deadlocks that we discuss in Chapter 5.

Whenever a process is interrupted, whether because of an I/O request or another type of interrupt, its processor updates the corresponding entry in the process list and finds another process to run. This means that the processors are kept quite busy. But it also means that any given job or task may be executed by several different processors during its run time. And because each processor has access to all I/O devices and can reference any storage unit, there are more conflicts as several processors try to access the same resource at the same time.

This presents the obvious need for algorithms to resolve conflicts among processors.

Process Synchronization Software

The success of **process synchronization** hinges on the capability of the operating system to make a resource unavailable to other processes while it is being used by one of them. These "resources" can include scanners and other I/O devices, a location in storage, or a data file, to name a few. In essence, the resource that's being used must be locked away from other processes until it is released. Only then is a waiting process allowed to use the resource. This is where synchronization is critical. A mistake could leave a job waiting indefinitely, causing starvation, or if it's a key resource, cause a deadlock. Both are described in Chapter 5.

It is the same thing that can happen in a crowded ice cream shop. Customers take a number to be served. The number machine on the wall shows the number of the person to be served next, and the clerks push a button to increment the displayed number when they begin attending to a new customer. But what happens when there is no synchronization between serving the customers and changing the number? Chaos. This is the case of the missed waiting customer.

Let's say your number is 75. Clerk 1 is waiting on Customer 73 and Clerk 2 is waiting on Customer 74. The sign on the wall says "Now Serving #74," and you're ready with your order. Clerk 2 finishes with Customer 74 and pushes the button so that the sign says "Now Serving #75." But just then, that clerk is called to the telephone and leaves the building due to an emergency, never to return (an interrupt). Meanwhile, Clerk 1 pushes the button and proceeds to wait on Customer 76—and you've missed your turn! If you speak up quickly, you can correct the mistake gracefully; but when it happens in a computer system, the outcome isn't as easily remedied.

Consider the scenario in which Processor 1 and Processor 2 finish with their current jobs at the same time. To run the next job, each processor must:

1. Consult the list of jobs to see which one should be run next.
2. Retrieve the job for execution.
3. Increment the READY list to the next job.
4. Execute it.

Both go to the READY list to select a job. Processor 1 sees that Job 74 is the next job to be run and goes to retrieve it. A moment later, Processor 2 also selects Job 74 and goes to retrieve it. Shortly thereafter, Processor 1, having retrieved Job 74, returns to the READY list and increments it, moving Job 75 to the top. A moment later Processor 2 returns; it has also retrieved Job 74 and is ready to process it, so it increments the READY list and now Job 76 is moved to the top and becomes the next job in line to be processed. Job 75 has become the missed waiting customer and will never be processed, while Job 74 is being processed twice—both represent an unacceptable state of affairs.

There are several other places where this problem can occur: memory and page allocation tables, I/O tables, application databases, and any shared resource.

Obviously, this situation calls for synchronization. Several synchronization mechanisms are available to provide cooperation and communication among processes. The common element in all synchronization schemes is to allow a process to finish work on a critical part of the program before other processes have access to it. This is applicable both to multiprocessors and to two or more processes in a single-processor (time-shared) system. It is called a **critical region** because it is a critical section and its execution must be handled as a unit. As we've seen, the processes within a critical region can't be interleaved without threatening the integrity of the operation.

The lock-and-key technique is conceptually the same one that's used to lock databases, as discussed in Chapter 5. It allows different users to access the same database without causing a deadlock.

Synchronization is sometimes implemented as a lock-and-key arrangement: Before a process can work on a critical region, it must get the key. And once it has the key, all other processes are locked out until it finishes, unlocks the entry to the critical region, and returns the key so that another process can get the key and begin work.

This sequence consists of two actions: (1) the process must first see if the key is available and (2) if it is available, the process must pick it up and put it in the lock to make it unavailable to all other processes. For this scheme to work, both actions must be performed in a single machine cycle; otherwise it is conceivable that while the first process is ready to pick up the key, another one would find the key available and prepare to pick up the key—and each could block the other from proceeding any further.

Several locking mechanisms have been developed, including test-and-set, WAIT and SIGNAL, and semaphores.

Test-and-Set

Test-and-set is a single, indivisible machine instruction known simply as **TS** and was introduced by IBM for its early multiprocessing computers. In a single machine cycle, it tests to see if the key is available and, if it is, sets it to unavailable.

The actual key is a single bit in a storage location that can contain a 0 (if it's free) or a 1 (if busy). We can consider TS to be a function subprogram that has one parameter

(the storage location) and returns one value (the condition code: busy/free), with the notable feature that it takes only one machine cycle.

Therefore, a process (Process 1) tests the condition code using the TS instruction before entering a critical region. If no other process is in this critical region, then Process 1 is allowed to proceed and the condition code is changed from 0 to 1. Later, when Process 1 exits the critical region, the condition code is reset to 0 so another process is allowed to enter. On the other hand, if Process 1 finds a busy condition code when it arrives, then it's placed in a waiting loop where it continues to test the condition code—when it's free, it's allowed to enter.

Although it's a simple procedure to implement, and it works well for a small number of processes, test-and-set has two major drawbacks:

- First, when many processes are waiting to enter a critical region, starvation can occur because the processes gain access in an arbitrary fashion. Unless a first-come, first-served policy is set up, some processes could be favored over others.
- Second, the waiting processes remain in unproductive, resource-consuming wait loops, requiring context switching, because the processes repeatedly check for the key. This is known as **busy waiting**—which not only consumes valuable processor time, but also relies on the competing processes to test the key—something that is best handled by the operating system or the hardware.

WAIT and SIGNAL

WAIT and SIGNAL is a modification of test-and-set that's designed to remove busy waiting. Two new operations, WAIT and SIGNAL, are mutually exclusive and become part of the process scheduler's set of operations.

WAIT is activated when the process encounters a busy condition code. WAIT sets the process's process control block (PCB) to the blocked state and links it to the queue of processes waiting to enter this particular critical region. The Process Scheduler then selects another process for execution. SIGNAL is activated when a process exits the critical region and the condition code is set to "free." It checks the queue of processes waiting to enter this critical region and selects one, setting it to the READY state. Eventually the Process Scheduler will choose this process for running. The addition of the operations WAIT and SIGNAL frees the processes from the busy waiting dilemma and returns control to the operating system, which can then run other jobs while the waiting processes are idle (WAIT).

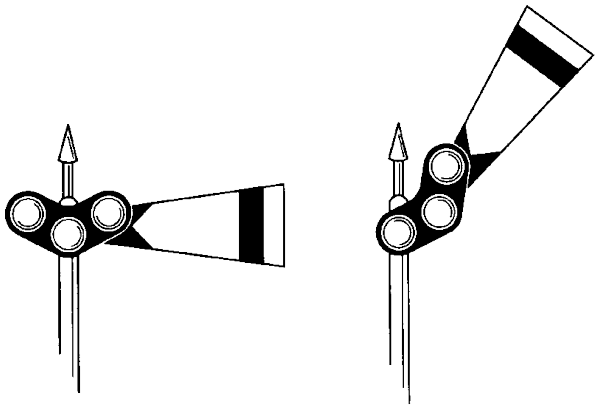
Semaphores

A **semaphore** is a non-negative integer variable that can be used as a binary signal, a flag. One of the most historically significant semaphores was the signaling device,

shown in Figure 6.4, used by railroads to indicate whether a section of track was clear. When the arm of the semaphore was raised, the track was clear and the train was allowed to proceed. When the arm was lowered, the track was busy and the train had to wait until the arm was raised. It had only two positions: up or down (on or off).

(figure 6.4)

The semaphore used by railroads indicates whether your train can proceed. When it's lowered (a), another train is approaching and your train must stop to wait for it to pass. If it is raised (b), your train can continue.



(a) Stop (b) All Clear

In an operating system, a semaphore is set to either zero or one to perform a similar function: It signals if and when a resource is free and can be used by a process. Dutch computer scientist Edsger Dijkstra (1965) introduced two operations to overcome the process synchronization problem we've discussed. Dijkstra called them P and V, and that's how they're known today. The P stands for the Dutch word *proberen* (to test) and the V stands for *verhogen* (to increment). The P and V operations do just that: They test and increment.

Here's how semaphores work: If we let *s* be a semaphore variable, then the V operation on *s* is simply to increment *s* by 1. The action can be stated as:

$$V(s): s := s + 1$$

This in turn necessitates a fetch (to get the current value of *s*), increment (to add one to *s*), and store sequence. Like the test-and-set operation, the increment operation must be performed as a single indivisible action to avoid race conditions. And that means that *s* cannot be accessed by any other process during the operation.

The operation P on *s* is to test the value of *s*, and if it's not 0, to decrement it by 1. The action can be stated as:

$$P(s): \text{If } s > 0 \text{ then } s := s - 1$$

This involves a test, fetch, decrement, and store sequence. Again, this sequence must be performed as an indivisible action in a single machine cycle or be arranged so that the process cannot take action until the operation (test or increment) is finished.

The operations to test or increment are executed by the operating system in response to calls issued by any one process naming a semaphore as parameter (this alleviates the

process from having control). If $s = 0$, it means that the critical region is busy and the process calling on the test operation must wait until the operation can be executed; that's not until $s > 0$.

In the example shown in Table 6.3, P3 is placed in the WAIT state (for the semaphore) on State 4. Also shown in Table 6.3, for States 6 and 8, when a process exits the critical region, the value of *s* is reset to 1, indicating that the critical region is free. This, in turn, triggers the awakening of one of the blocked processes, its entry into the critical region, and the resetting of *s* to 0. In State 7, P1 and P2 are not trying to do processing in that critical region, and P4 is still blocked.

State Number	Calling Process	Operation	Running in Critical Region	Results Blocked on *s*	Value of *s*
0					1
1	P1	test(*s*)	P1		0
2	P1	increment(*s*)			1
3	P2	test(*s*)	P2		0
4	P3	test(*s*)	P2	P3	0
5	P4	test(*s*)	P2	P3, P4	0
6	P2	increment(*s*)	P3	P4	0
7			P3	P4	0
8	P3	increment(*s*)	P4		0
9	P4	increment(*s*)			1

(table 6.3)

The sequence of states for four processes (P1, P2, P3, P4) calling test and increment (P and V) operations on the binary semaphore s. (Note: The value of the semaphore before the operation is shown on the line preceding the operation. The current value is on the same line.)

After State 5 of Table 6.3, the longest waiting process, P3, was the one selected to enter the critical region, but that isn't necessarily the case unless the system is using a first-in, first-out selection policy. In fact, the choice of which job will be processed next depends on the algorithm used by this portion of the Process Scheduler.

As you can see from Table 6.3, test and increment operations on semaphore *s* enforce the concept of mutual exclusion, which is necessary to avoid having two operations in a race condition. The name traditionally given to this semaphore in the literature is **mutex,** for MUTual EXclusion. So the operations become:

```
test(mutex):      if mutex > 0 then mutex: = mutex - 1
increment(mutex): mutex: = mutex + 1
```

In Chapter 5 we talk about the requirement for mutual exclusion when several jobs are trying to access the same shared physical resources. The concept is the same here,

but we have several processes trying to access the same shared critical region. The procedure can generalize to semaphores having values greater than 0 and 1.

Thus far we've looked at the problem of mutual exclusion presented by interacting parallel processes using the same shared data at different rates of execution. This can apply to several processes on more than one processor, or interacting (codependent) processes on a single processor. In this case, the concept of a critical region becomes necessary because it ensures that parallel processes will modify shared data only while in the critical region.

In sequential computations mutual exclusion is achieved automatically because each operation is handled in order, one at a time. However, in parallel computations, the order of execution can change, so mutual exclusion must be explicitly stated and maintained. In fact, the entire premise of parallel processes hinges on the requirement that all operations on common variables consistently exclude one another over time.

Process Cooperation

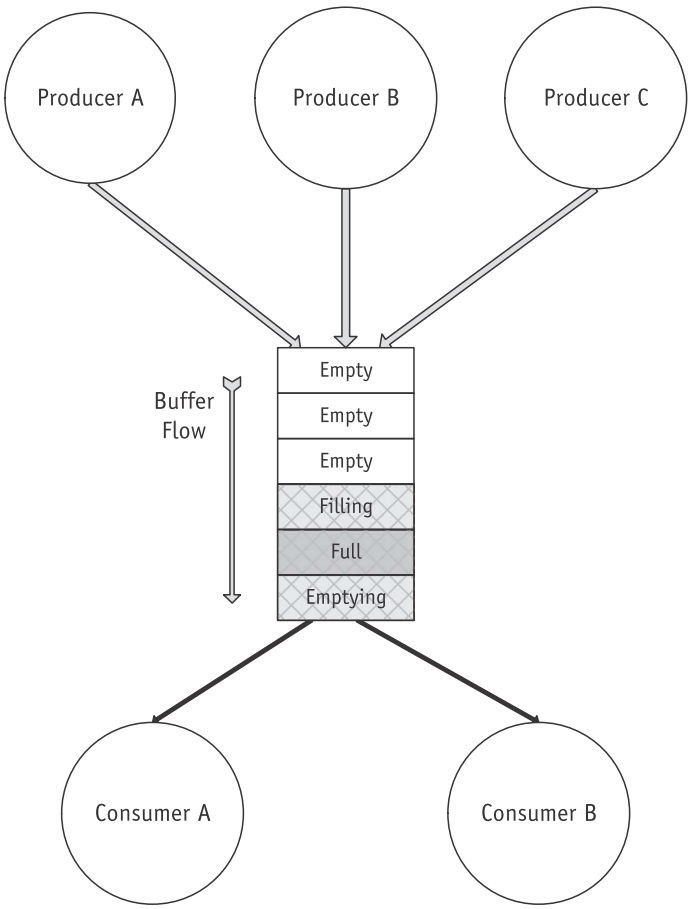
There are occasions when several processes work directly together to complete a common task. Two famous examples are the problems of producers and consumers, and of readers and writers. Each case requires both mutual exclusion and synchronization, and each is implemented by using semaphores.

Producers and Consumers

The classic problem of **producers and consumers** concerns one or more processes that produce data that one or more process consumes later. They do so by using a single buffer. Let's begin with the case with one producer and one consumer, although current versions would almost certainly apply to systems with multiple producers and consumers, as shown in Figure 6.5.

Let's return for a moment to the fast-food framework at the beginning of this chapter because the synchronization between one producer process and one consumer process (the cook and the bagger) represents a significant issue when it's applied to operating systems. The cook *produces* hamburgers that are sent to the bagger, who *consumes* them. Both processors have access to one common area, the hamburger bin, which can hold only a finite number of hamburgers (the bin is essentially a buffer). The bin is a necessary storage area because the speed at which hamburgers are produced is rarely exactly the same speed at which they are consumed.

Problems arise at two extremes: when the producer attempts to add to a bin that's already full (as happens when the consumer stops all withdrawals from the bin), and



(figure 6.5)

Three producers fill one buffer that is emptied by two consumers.

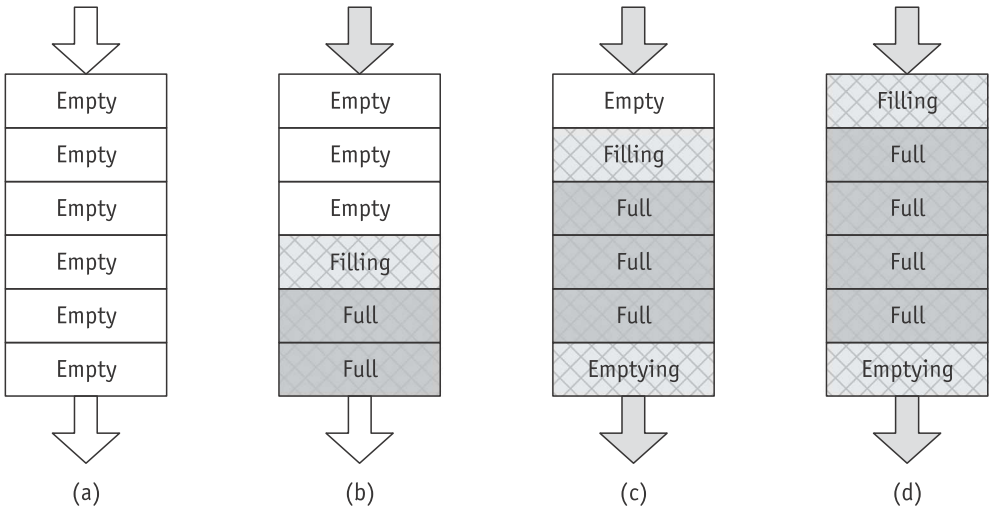
when the consumer attempts to make a withdrawal from a bin that's empty (such as when the bagger tries to take a hamburger that hasn't been deposited in the bin yet). In real life, the people watch the bin; if it's empty or too full, they quickly recognize and resolve the problem. However, in a computer system, such resolution is not so easy.

Consider the case of the prolific CPU that can generate output data much faster than a printer can print it. Therefore, to accommodate the two different speeds, we need a buffer where the producer can temporarily store data that can be retrieved by the consumer at a slower speed, freeing the CPU from having to wait unnecessarily. This buffer can be in many states, from empty to full, as shown in Figure 6.6.

The essence of the problem is that the system must make sure that the producer won't try to add data to a full buffer, and that the consumer won't try to make withdrawals from an empty buffer. Because the buffer can hold only a finite amount of data, the synchronization process must delay the producer from generating more data when the buffer is full. It must also be prepared to delay the consumer from retrieving data when the buffer is empty. This task can be implemented by two counting semaphores—one to indicate the number of *full* positions in the buffer and the other to indicate the

(figure 6.6)

Four snapshots of a single buffer in four states from completely empty (a) to almost full (d)

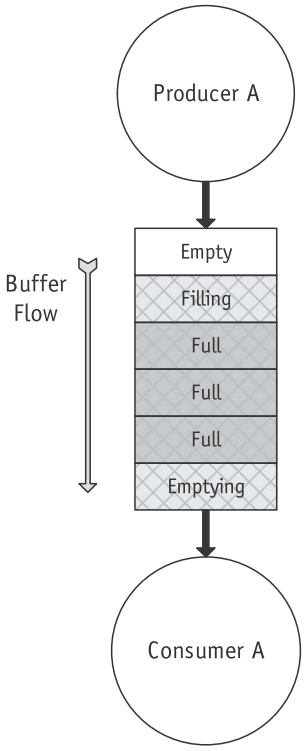


number of *empty* positions in the buffer. A third semaphore, mutex, ensures mutual exclusion between processes to prevent race conditions.

Notice that when there is only one producer and one consumer, this classic problem becomes a simple case of first in, first out (FIFO), as shown in Figure 6.7. However, when there are several producers and consumers, a deadlock can occur if they are all paused as they wait to be notified that they can resume their respective deposits and withdrawals.

(figure 6.7)

Typical system with one producer, one consumer, and a single buffer



Readers and Writers

The problem of **readers and writers** was first formulated by Courtois, Heymans, and Parnas (1971) and arises when two types of processes need to access a shared resource such as a file or database. The authors called these processes readers and writers.

An airline reservation system is a good example. The readers are those who want flight information. They're called readers because they only read the existing data; they don't modify it. And because none of them is changing the database, the system can allow many readers to be active at the same time—there's no need to enforce mutual exclusion among them.

The writers are those who are making reservations on a particular flight. Unlike the readers, the writers must be carefully accommodated for each flight because they are modifying existing data in the database. The system can't allow someone to be writing while someone else is reading or writing to the exact same file. Therefore, it must enforce mutual exclusion for any and all writers. Of course, the system must be fair when it enforces its policy to avoid indefinite postponement of readers or writers.

In the original paper, Courtois, Heymans, and Parnas offered two solutions using P and V operations.

- The first gives priority to readers over writers so readers are kept waiting only if a writer is actually modifying the data. However, if there is a continuous stream of readers, this policy results in writer starvation.
- The second policy gives priority to the writers. In this case, as soon as a writer arrives, any readers that are already active are allowed to finish processing, but all additional readers are put on hold until the writer is done. Obviously, if a continuous stream of writers is present, this policy results in reader starvation.
- Either scenario is unacceptable.

To prevent either type of starvation, Hoare (1974) proposed the following combination priority policy: When a writer is finished, any and all readers who are waiting or on hold are allowed to read. Then, when that group of readers is finished, the writer who is on hold can begin; any *new* readers who arrive in the meantime aren't allowed to start until the writer is finished.

The state of the system can be summarized by four counters initialized to 0:

- Number of readers who have *requested* a resource and haven't yet released it (R1 = 0)
- Number of readers who are *using* a resource and haven't yet released it (R2 = 0)
- Number of writers who have *requested* a resource and haven't yet released it (W1 = 0)
- Number of writers who are *using* a resource and haven't yet released it (W2 = 0)

This can be implemented using two semaphores to ensure mutual exclusion between readers and writers. A resource can be given to all readers, provided that no writers

are processing (W2 = 0). A resource can be given to a writer, provided that no readers are reading (R2 = 0) and no writers are writing (W2 = 0).

Readers must always call two procedures: the first checks whether the resources can be immediately granted for reading; and then, when the resource is released, the second checks to see if there are any writers waiting. The same holds true for writers. The first procedure must determine if the resource can be immediately granted for writing, and then, upon releasing the resource, the second procedure will find out if any readers are waiting.

Concurrent Programming

Until now, we've looked at multiprocessing as several jobs executing at the same time on a single processor (which interacts with I/O processors, for example) or on multiprocessors. Multiprocessing can also refer to one job using several processors to execute sets of instructions in parallel. The concept isn't new, but it requires a programming language and a computer system that can support this type of construct. This type of system is referred to as a **concurrent processing** system, and it can generally perform data level parallelism and instruction (or task) level parallelism.

Frances E. Allen (1932–)

Fran Allen, mathematician and teacher, joined IBM intending only to pay off student loans; however, she stayed for 45 years as an innovator in sophisticated systems analysis, parallel processing, and code optimization. Her techniques are still reflected in programming language compilers. Allen's last big project for IBM was the Parallel Translator, where she applied her extensive experience with interprocedural flow analysis to produce new algorithms for extracting parallelism from sequential code. Allen was the first woman named an IBM Fellow (1989) and was the first female to win the ACM Turing Award (2006). As of this writing, she continues to advise IBM on a number of projects, including its Blue Gene supercomputer.

For more information:
http://amturing.acm.org/award_winners/allen_1012327.cfm

Won the Association for Computing Machinery Turing Award: "For pioneering contributions to the theory and practice of optimizing compiler techniques that laid the foundation for modern optimizing compilers and automatic parallel execution."

Courtesy of International Business Machines Corporation, © International Business Machines Corporation.

In general, parallel systems can be put into two broad categories: **data level parallelism (DLP),** which refers to systems that can perform on one or more streams or elements of data, and **instruction (or task) level parallelism (ILP)**, which refers to systems that can perform multiple instructions in parallel.

Michael Flynn (1972) published what's now known as Flynn's taxonomy, shown in Table 6.4, describing machine structures that fall into four main classifications of parallel construction with combinations of single/multiple instruction and single/multiple data. Each category presents different opportunities for parallelism.

	Number of Instructions	
	Single Instruction	**Multiple Instructions**
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

(table 6.4)

The four classifications of Flynn's Taxonomy for machine structures

Each category presents different opportunities for parallelism.

- The single instruction, single data (SISD) classification is represented by systems with a single processor and a single stream of data which presents few if any opportunities for parallelism.
- The multiple instructions, single data (MISD) classification includes systems with multiple processors (which might allow some level of parallelism) and a single stream of data. Configurations such as this might allow instruction level parallelism but little or no data level parallelism without additional software assistance.
- The single instruction, multiple data (SIMD) classification is represented by systems with a single processor and a multiple data streams.
- The multiple instructions, multiple data (MIMD) classification is represented by systems with a multiple processors and a multiple data streams. These systems may allow the most creative use of both instruction level parallelism and data level parallelism.

Amdahl's Law

Regardless of a system's physical configuration, the result of parallelism is more complex than initially thought. Gene Amdahl proposed in 1967 that the limits of parallel processors could be calculated, as shown in Figure 6.8.

In its simplest form, Amdahl's Law maintains that if a given program would take one hour to run to completion with one processor, but 80 percent of the program had to run sequentially (and therefore could not be sped up by applying additional processors to it), then the resulting program could run only a maximum of 20 percent faster. Interestingly, Amdahl used his work to promote the use of single processor systems even as parallel processors were becoming available.

(figure 6.8)

Amdahl's Law. Notice that all four graphs level off and there is no speed difference even though the number of processors increased from 2,048 to 65,536 (Amdahl, 1967).

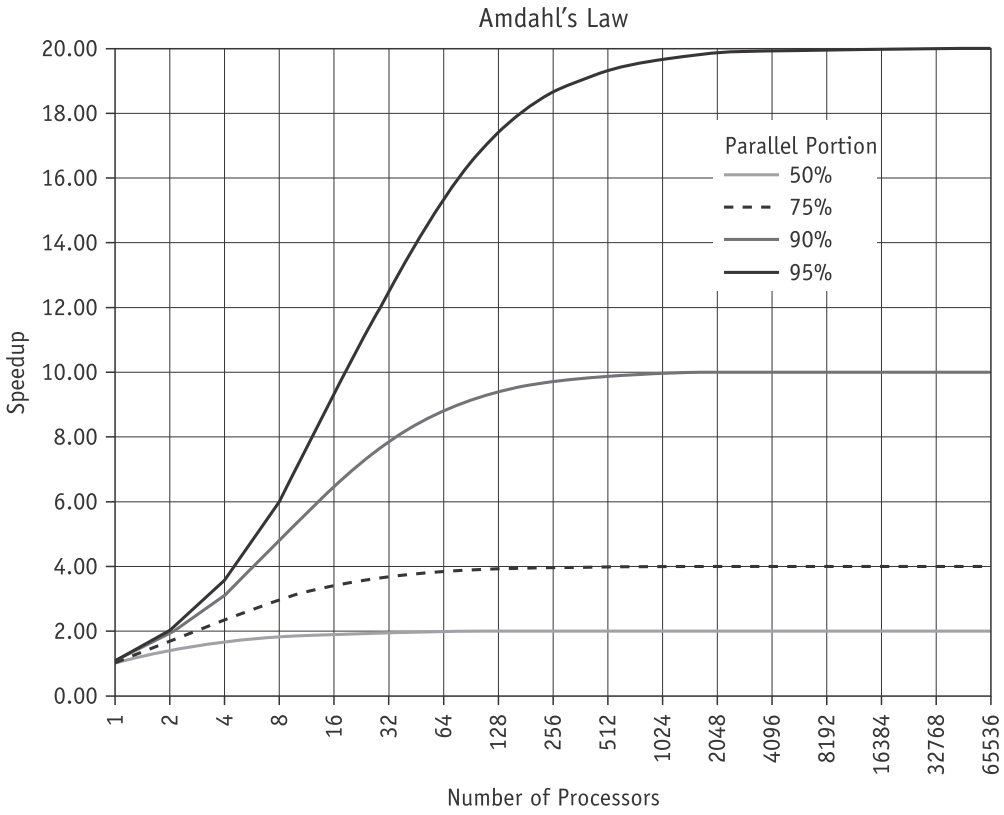


Photo Source: Wikipedia among many others http://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:AmdahlsLaw.png.

Order of Operations

Before we delve more into parallelism, let's explore a fundamental concept that applies to all calculations. To understand the importance of this, try solving the following equation twice, once while resolving each calculation from the left, and then do the same thing from the right.

$$Z = 10 * 4 - 2$$

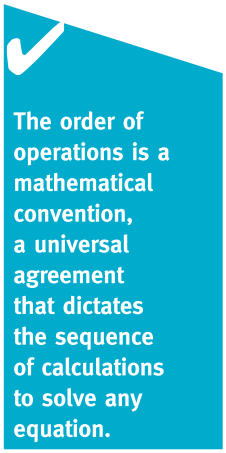
If we start from the left and do the multiplications first (10 * 4 = 40) and then the subtraction, we find that Z = 38. However, if we start from the right and do the subtraction first (4 – 2 = 2) and then do the multiplication, we find that Z = 20.

Therefore, to assure that every equation results in the exactly the same answer every time, mathematical convention states that each operation must be performed in a certain sequence or **order of operations** (also called precedence of operations or rules of precedence). These rules state that to solve an equation, all arithmetic calculations are performed *from the left* and in the following order:

- First, all calculations in parentheses are performed.
- Second, all exponents are calculated.

- Third, all multiplications and divisions are performed (they're resolved from the left whether they are divisions or multiplications—divisions do not wait until all multiplications are solved).
- Fourth, all additions and subtractions are performed (they're grouped together and then resolved from the left).

Each step in the equation is evaluated from left to right (because, if you were to perform the calculations in some other order, you would run the risk of finding the incorrect answer). In equations that contain a combination of multiplications and divisions, each is performed from the left whether they require division or multiplication. That is, we don't perform all multiplications first, followed by all divisions. Likewise, all subtractions and additions are performed from the left, even if subtractions are performed before any additions that appear to their right.



The order of operations is a mathematical convention, a universal agreement that dictates the sequence of calculations to solve any equation.

For example, if the equation is $Z = A / B + C * D$,

1. the division is performed first $(A / B = M)$,
2. the multiplication is performed second $(C * D = N)$, and
3. only then is the final sum performed $(Z = M + N)$.

On the other hand, if the equation includes one or more parentheses, such as $Z = A / (B + C) * D$, then

1. the parenthetical phrase is performed first $(B + C = M)$
2. the division is performed $(A / M = N)$ *next*.
3. Finally the multiplication is performed $(Z = N * D)$.

For many computational purposes and simple calculations, serial processing is sufficient; it's easy to implement and fast enough for most users. However, arithmetic expressions can be processed in a very different and more efficient way if we use a language that allows for concurrent processing.

Returning to our first equation, $Z = (A / B + C * D)$, notice that in this case the division $(A / B = M)$ can take place at the same time as the multiplication $(C * D = N)$. If the system had two available CPUs, each calculation could be performed simultaneously—each by a different CPU. Then (and only then) one CPU would add together the results of the two previous calculations $(Z = M + N)$.

Suppose the equation to be solved is more complex than our previous example. (Note the double asterisk indicates an exponent, and the two operations in parentheses need to be performed before all others so the exponential value can be found.)

$$Z = 10 - A / B + C (D + E) ** (F - G)$$

Table 6.5 shows the steps to compute it.

(table 6.5)

The sequential computation of the expression requires several steps. (In this example, there are six steps, but each step, such as the last one, may involve more than one machine operation.)

Step No.	Operation	Result
1	(D + E) = M	Store sum in M
2	(F – G) = N	Store difference in N
		Now the equation is Z = 10 – A/B + C(M^N)
3	(M) ** (N) = P	Store power in P
		Now the equation is Z = 10 – A/B + C(P)
4	A / B = Q	Store quotient in Q
		Now the equation is Z = 10 – Q + C(P)
5	C * P = R	Store product in R
		Now the equation is Z = 10 – Q + R
6	10 – Q + R = Z	Store result in Z

Applications of Concurrent Programming

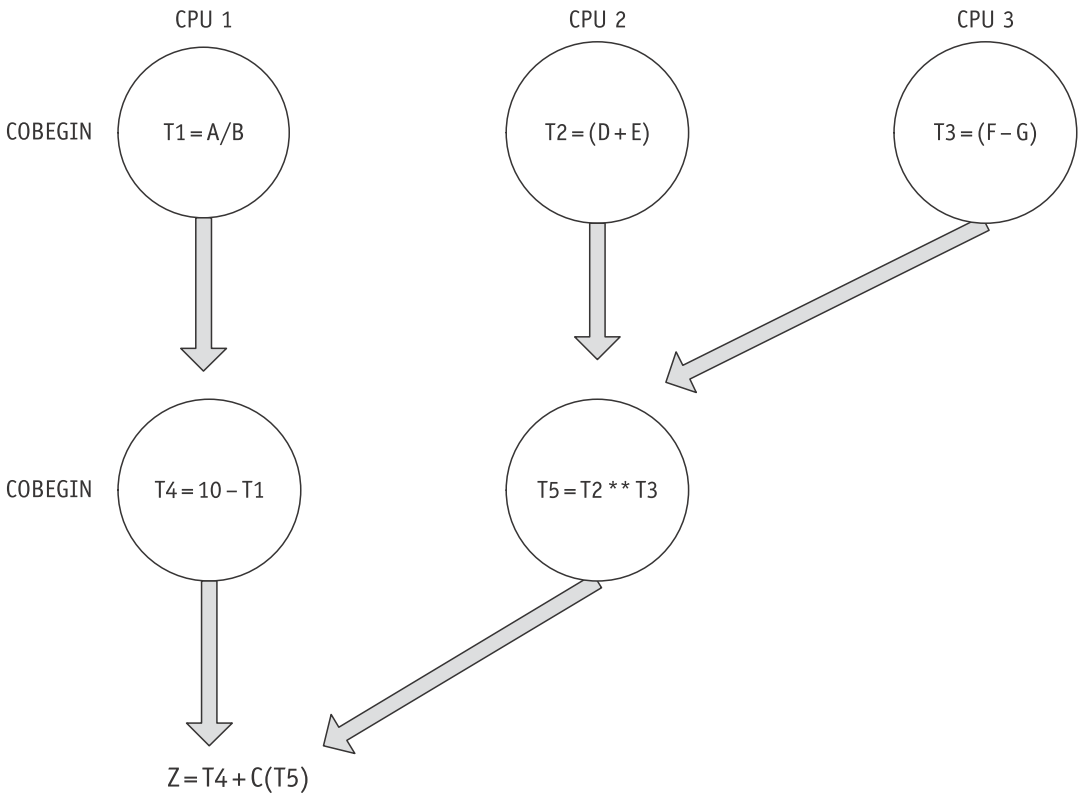
Can we evaluate this equation faster if we use multiple processors? Let's introduce two terms—**COBEGIN** and **COEND**—that identify the instructions that can be processed concurrently (assuming that a sufficient number of CPUs are available). When we rewrite our expression to take advantage of concurrent processing, and place the results in three temporary storage locations (T1, T2, T3, and so on), it can look like this:

```
COBEGIN
     T1 = A / B
     T2 = (D + E)
     T3 = (F - G)
COEND

COBEGIN
     T4 = 10 - T1
     T5 = T2 ** T3
COEND

     Z = T4 + C(T5)
```

As shown in Table 6.5, to solve *Z = 10 – A / B + C (D + E) ** (F – G)*, the first three operations can be done at the same time if our computer system has at least three processors. The next two operations are done at the same time, and the last expression is performed serially with the results of the first two steps, as shown in Figure 6.9.



(figure 6.9)

Three CPUs can perform the six-step equation in three steps.

With this system, we've increased the computation speed, but we've also increased the complexity of the programming language and the hardware (both machinery and communication among machines). In fact, we've also placed a large burden on the programmer—to explicitly state which instructions can be executed in parallel. This is **explicit parallelism**.

When the **compiler** automatically detects the instructions that can be performed in parallel, it is called **implicit parallelism**, a field that benefited from the work of Turing Award winner Fran Allen.

It is the compiler that's used in multiprocessor systems (or even a massively multiprocessing system such as the one shown in Figure 6.10), that translates the algebraic expression into separate instructions and decides which steps can be performed in parallel and which in serial mode.

For example, the equation $Y = A + B * C + D$ could be rearranged by the compiler as $A + D + B * C$ so that two operations $A + D$ and $B * C$ would be done in parallel, leaving the final addition to be calculated last.

As shown in the four cases that follow, concurrent processing can also dramatically reduce the complexity of working with array operations within loops, of performing

(figure 6.10)

The IBM Sequoia supercomputer, deemed the fastest on earth in 2012 by TOPS (http://www.top500.org), features 98,304 IBM 16-core processors, which equals more than 1.5 million processing cores. It has 1.6 petabytes of RAM and runs Linux. Courtesy of Lawrence Livermore National Laboratory

matrix multiplication, of conducting parallel searches in databases, and of sorting or merging files. Some of these systems use parallel processors that execute the same type of tasks.

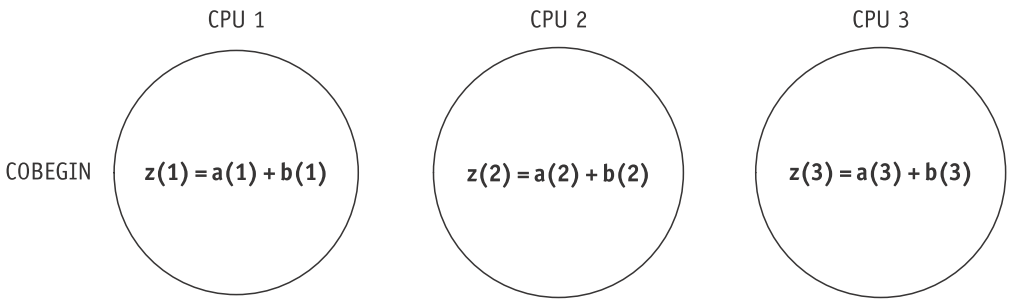
Case 1: Array Operations

Let's say we need to perform an instruction on the objects in an array (an array is a collection of objects that have the same attributes so each object can be addressed individually).To do so, the code might say this:

```
for(j = 1; j <= 3; j++)
z(j) = a(j) + b(j);
```

These two instructions tell the Processor Manager to perform one identical operation *(a + b)* on three different objects: z(1) = a(1) + b(1), z(2) = a(2) + b(2), z(3) = a(3) + b(3).

Because the operation will be performed on three different objects, we can use three different processors to perform the instruction in one step as shown in Figure 6.11. Likewise, if the instruction is to be performed on 20 objects, and if there are 20 CPUs available, they can all be performed in one step, as well.



(figure 6.11)

Using CPUs and the COBEGIN command, three instructions in one array can be performed in a single step.

Case 2: Matrix Multiplication

Matrix multiplication requires many multiplication and addition operations that can take place concurrently, such as this equation: Matrix C = Matrix 1 * Matrix 2.

$$\underset{\text{Matrix C}}{\begin{bmatrix} z & y & x \\ w & v & u \\ t & s & r \end{bmatrix}} = \underset{\text{Matrix 1}}{\begin{bmatrix} A & B & C \\ D & E & F \end{bmatrix}} * \underset{\text{Matrix 2}}{\begin{bmatrix} K & L \\ M & N \\ O & P \end{bmatrix}}$$

To find z in Matrix C, you multiply each element from the first *column* of Matrix 1 by the corresponding element in the first *row* of Matrix 2, and then add the products. Therefore, one calculation is this: z = (A * K) + (D * L). Likewise, x = (C * K) + (F * L) and s = (B * O) + (E * P).

Using one processor, the answer to this equation can be computed in 27 steps. By multiplying several elements of the first row of Matrix 1 by corresponding elements in Matrix 2, three processors could cooperate to resolve this equation in fewer steps. The number of products that could be computed at the same time would depend on the number of processors available. With two processors, it takes only 18 steps to perform the calculations in parallel. With three, it would require even fewer. Notice that concurrent processing does not necessarily cut processing activity in direct proportion to the number of processors available. In this example, by doubling the number of processors from one to two, the number of steps was reduced by one-third—not by one-half.

Case 3: Searching Databases

Database searching is a common non-mathematical application for concurrent processing systems. For example, if a word is sought from a dictionary database or a part number from an inventory listing, the entire file can be split into discrete sections, with one processor allocated to each section. This results in a significant reduction in search time. Once the item is found, all processors can be deallocated and set to work on the

next task. Even if the item sought is in the last record of the database, a concurrent search is faster than if a single processor were allocated to search the database.

Case 4: Sorting and Merging Files

The task of sorting a file or merging several sorted files can benefit from multiple processors. By dividing a large file into sections, each with its own processor, every section can be sorted at the same time. Then pairs of sections can be merged together until the entire file is whole again—and sorted. It's very similar to what you and a friend could do if you each held a portion of a deck of cards. Each of you (playing the part of a processor) could sort your own cards before you'd merge the two piles into one sorted deck.

Threads and Concurrent Programming

So far we have considered the cooperation and synchronization of traditional processes, also known as heavyweight processes, which have the following characteristics:

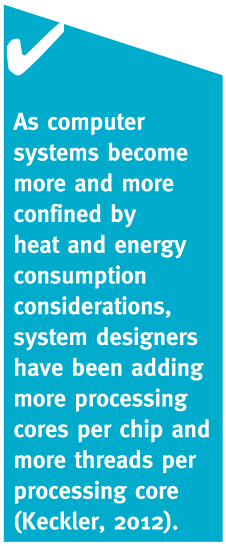
- They pass through several states from their initial entry into the computer system to their completion: ready, running, waiting, delayed, and blocked.
- They require space in main memory where they reside during their execution.
- From time to time they require other resources, such as data.

As we have seen in Chapters 3 and 4, these processes are often swapped between main memory and secondary storage during their execution to accommodate multiprogramming and to take advantage of virtual memory. Every time a swap occurs, overhead increases because of all the information that has to be saved.

To minimize this overhead time, most current operating systems support the implementation of threads, or lightweight processes, which have become part of numerous application packages. Threads are supported at both the kernel and at the user level; they can be managed by either the operating system or the application that created them.

A thread, discussed in Chapter 4, is a smaller unit within a process, which can be scheduled and executed. Threads share the same resources as the process that created them, which now becomes a more passive element because the thread is the unit that uses the CPU and is scheduled for execution. Processes might have from one to several active threads, which can be created, scheduled, and synchronized more efficiently because the amount of information needed is reduced. When running a process with multiple threads in a computer system with a single CPU, the processor switches very quickly from one thread to another, giving the impression that the threads are executing in parallel. However, it is only in systems with multiple CPUs that the multiple threads in a process are actually executed in parallel.

Each active thread in a process has its own processor registers, program counter, stack, and status, but each thread shares the data area and the resources allocated to its process. Each thread has its own program counter, which is used to store variables dynamically created by a process. For example, function calls in C might create variables that are local to the function. These variables are stored in the stack when the function is invoked and are destroyed when the function is exited. Since threads within a process share the same space and resources, they can communicate more efficiently, increasing processing performance.



Consider how a Web server can improve performance and interactivity by using threads. When a Web server receives requests for images or pages, it serves each request with a different thread. For example, the process that receives all the requests may have one thread that accepts these requests and creates a new separate thread for each request received. This new thread retrieves the required information and sends it to the remote client. While this thread is doing its task, the original thread is free to accept more requests. Web servers are multiprocessor systems that allow for the concurrent completion of several requests, thus improving throughput and response time. Instead of creating and destroying threads to serve incoming requests, which would increase the overhead, Web servers keep a pool of threads that can be assigned to those requests. After a thread has completed its task, instead of being destroyed, it is returned to the pool to be assigned to another request.

Two Concurrent Programming Languages

Early programming languages did not support the creation of threads or the existence of concurrent processes. Typically, they gave programmers the possibility of creating a single process or thread of control. The Ada programming language was one of the first to do so in the late 1970s.

Ada

Ada 2012 is the International Standards Organization (ISO) standard and replaces Ada 2005. The scope of this revision is guided by a document issued to the committee charged with drafting the revised standard, the Ada Rapporteur Group (ARG). The essence is that the ARG was asked to pay particular attention to both object-oriented programming and to add real-time elements with a strong emphasis on real-time and high integrity features (Barnes, 2012). Specifically, ARGs charter was to:

1. Make improvements to maintain or improve Ada's advantages, especially in those user domains where safety and criticality are prime concerns, including improving the capabilities of Ada to take advantage of multicore and multithreaded architectures.

2. Make improvements to remedy shortcomings in Ada, such as the safety, use, and functionality of access types and dynamic storage management.

The U.S. Department of Defense (DoD) first commissioned the creation of the Ada programming language to support concurrent processing and the embedded computer systems found on jet aircraft, ship controls, and spacecraft, to name just a few. These types of systems must typically work with parallel processors, real-time constraints, fail-safe execution, and nonstandard input and output devices. It took ten years to complete; Ada was made available to the public in 1980.

It's named after Augusta Ada Byron, the Countess of Lovelace and a skilled mathematician who is regarded by some as the world's first programmer for her work on Charles Babbage's Analytical Engine in the 1830s. The Analytical Engine was an early prototype of a computer (Barnes, 1980). Notably, the mathematician was the daughter of renowned poet Lord Byron.

From the beginning, the new language was designed to be modular so several programmers can work on sections of a large project independently of one another. This allows several contracting organizations to compile their sections separately and have the finished product assembled together when its components are complete. From the beginning of the project, the DoD realized that their new language would prosper only if there was a global authority that could stifle the growth of mutually incompatible subsets and supersets of the language by creating unique, enhanced versions of the standard compiler with extra "bells and whistles," thus making them incompatible with each other. Toward that end, the DoD officially trademarked the name "Ada" and the language is now an ISO standard. Changes to it are currently made under the guidance of an ISO/IEC committee that has included national representatives of many nations, including Belgium, Canada, France, Germany, Italy, Japan, Sweden, Switzerland, the United Kingdom, and the U.S. (Barnes, 2012).

Specific details of all individual changes are integrated to form a new version of the official Annotated Ada Reference Manual scheduled for publication by the end of 2012. For the latest details and a copy of the manual, see *http://www.ada-auth.org/standards*.

Java

Java, developed by Sun Microsystems, Inc., was designed as a universal software platform for Internet applications and has been widely adopted. Java was released in 1995 as the first popular software platform that allowed programmers to code an application with the capability to run on any computer. This type of universal software platform was an attempt to solve several issues: first, the high cost of developing

software applications for each of the many incompatible computer architectures available; second, the needs of distributed client-server environments; and third, the growth of the Internet and the Web, which added more complexity to program development.

Java uses both a compiler and an interpreter. The source code of a Java program is first compiled into an intermediate language called Java bytecodes, which are platform-independent. This means that one can compile a Java program on any computer that has a Java compiler, and the bytecodes can then be run on any computer that has a Java interpreter.

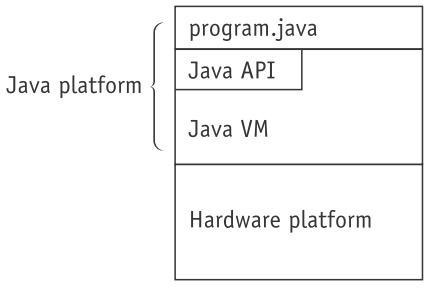
The interpreter is designed to fit in with the hardware and operating system specifications of the computer that will run the Java bytecodes. Its function is to parse and run each bytecode instruction on that computer.

This combination of compiler and interpreter makes it easy to distribute Java applications because they don't have to be rewritten to accommodate the characteristics of every computer system. Once the program has been compiled, it can be ported to, and run on, any system with a Java interpreter.

The Java Platform

Typically a computer platform contains the hardware and software where a program runs. The Java platform is a software-only platform that runs on top of other hardware-based platforms. It has two components: the Java Virtual Machine (Java VM), and the Java Application Programming Interface (Java API).

Java VM is the foundation for the Java platform and contains the Java interpreter, which runs the bytecodes provided by the compiler. Java VM sits on top of many different hardware-based platforms, as shown in Figure 6.12.



(figure 6.12)

A process used by the Java platform to shield a Java program from a computer's hardware.

The Java API is a collection of software modules that programmers can use in their applications. The Java API is grouped into libraries of related classes and interfaces. These libraries, also known as packages, provide well-tested objects ranging from basic data types to I/O functions, from network interfaces to graphical user interface kits.

The Java Language Environment

A strength of Java is that it allows programs to run on many different platforms, from powerful computers to telephones, without requiring customization for each platform.

Java was designed to make it easy for experienced programmers to learn. Its syntax is familiar because it looks and feels like C++. It is object-oriented, which means it takes advantage of modern software development methodologies and fits well into distributed client-server applications.

One of Java's features is that memory allocation is done at run time, unlike C and C++, where memory allocation is done at compilation time. Java's compiled code references memory via symbolic "handles" that are translated into real memory addresses at run time by the Java interpreter. This means that the memory allocation and referencing models are not visible to programmers, but are controlled entirely by the underlying run-time platform.

Because Java applications run in distributed environments, security is a very important built-in feature of the language and of the run-time system. It provides compile-time checking and run-time checking, which helps programmers create very reliable software. For example, while a Java program is executing, it can request particular classes to be loaded from anywhere in the network. In this case, all incoming code is checked by a verifier, which ensures that the code is correct and can run safely without putting the Java interpreter at risk.

With its sophisticated synchronization capabilities, Java supports multithreading at the language level. The language library provides the Thread class, and the run-time system provides monitor and condition lock primitives. The Thread class is a collection of methods used to start, run, stop, and check the status of a thread. Java's threads are preemptive and, depending on the platform on which the Java interpreter executes, can also be time-sliced.

When a programmer declares some methods within a class to be synchronized, they are not run concurrently. These synchronized methods are under the control of software that makes sure that variables remain in a consistent state. When a synchronized method begins to run, it is given a monitor for the current object, which does not allow any other synchronized method in that object to execute. The monitor is released when a synchronized method exits, which allows other synchronized methods within the same object to run.

Java technology continues to be popular with programmers for several reasons:

- It offers the capability of running a single program on various platforms without having to make any changes.
- It offers a robust set of features, such as run-time memory allocation, security, and multithreading.
- It is used for many Web and Internet applications, and it integrates well with browsers that can run Java applets with audio, video, and animation directly in a Web page.

Conclusion

Multiprocessing can occur in several configurations: in a single-processor system where interacting processes obtain control of the processor at different times, or in systems with multiple processors, where the work of each processor communicates and cooperates with the others and is synchronized by the Processor Manager. Three multiprocessing hardware systems are described in this chapter: master/slave, loosely coupled, and symmetric.

The success of any multiprocessing system depends on the ability of the system to synchronize its processes with the system's other resources. The concept of mutual exclusion helps keep the processes with the allocated resources from becoming deadlocked. Mutual exclusion is maintained with a series of techniques, including test-and-set, WAIT and SIGNAL, and semaphores: P, V, and mutex.

Hardware and software mechanisms are used to synchronize the many processes, but care must be taken to avoid the typical problems of synchronization: missed waiting customers, the synchronization of producers and consumers, and the mutual exclusion of readers and writers. Continuing innovations in concurrent processing, including threads and multicore processors, are fundamentally changing how operating systems use these new technologies. Research in this area is expected to grow significantly over time.

In the next chapter, we look at the module of the operating system that manages the printers, disk drives, tape drives, and terminals—the Device Manager.

Key Terms

busy waiting: a method by which processes, waiting for an event to occur, continuously test to see if the condition has changed and remain in unproductive, resource-consuming wait loops.

COBEGIN: command used with COEND to indicate to a multiprocessing compiler the beginning of a section where instructions can be processed concurrently.

COEND: command used with COBEGIN to indicate to a multiprocessing compiler the end of a section where instructions can be processed concurrently.

compiler: a computer program that translates programs from a high-level programming language (such as C or Ada) into machine language.

concurrent processing: execution by a single processor of a set of processes in such a way that they appear to be happening at the same time.

critical region: a part of a program that must complete execution before other processes can begin.

explicit parallelism: a type of concurrent programming that requires that the programmer explicitly state which instructions can be executed in parallel.

implicit parallelism: a type of concurrent programming in which the compiler automatically detects which instructions can be performed in parallel.

loosely coupled configuration: a multiprocessing configuration in which each processor has a copy of the operating system and controls its own resources.

master/slave: an asymmetric multiprocessing configuration consisting of a single processor system connected to "slave" processors each of which is managed by the primary "master" processor, which provides the scheduling functions and jobs.

multicore processor: a computer chip that contains more than a single central processing unit (CPU).

multiprocessing: when two or more processors share system resources that may include some or all of the following: the same main memory, I/O devices, and control program routines.

mutex: a condition that specifies that only one process may update (modify) a shared resource at a time to ensure correct operation and results.

order of operations: the algebraic convention that dictates the order in which elements of a formula are calculated.

parallel processing: the process of operating two or more CPUs executing instructions simultaneously.

pointer: an address or other indicator of location.

process synchronization: (1) the need for algorithms to resolve conflicts between processors in a multiprocessing environment; or (2) the need to ensure that events occur in the proper order even if they are carried out by several processes.

producers and consumers: a classic problem in which a process produces data that will be consumed, or used, by another process.

readers and writers: a problem that arises when two types of processes need to access a shared resource such as a file or a database.

semaphore: a type of shared data item that may contain either binary or nonnegative integer values and is used to provide mutual exclusion.

symmetric configuration: a multiprocessing configuration in which processor scheduling is decentralized and each processor is of the same type.

test-and-set (TS): an indivisible machine instruction, which is executed in a single machine cycle to determine whether the processor is available.

WAIT and SIGNAL: a modification of the test-and-set synchronization mechanism that's designed to remove busy waiting.

Interesting Searches

For more background on a few of the topics discussed in this chapter, begin a search with these terms.

- Multicore CPU Speed
- Multithreaded Processing
- Real-time Processing
- Cluster Computing
- Ada Programming Language

Exercises

Research Topics

A. Research current literature to identify a computer model with exceptional parallel processing ability. Identify the manufacturer, the maximum number of processors the computer uses, how fast the machine can perform calculations, and typical applications for it. Cite your sources. If your answer includes terms not used in this chapter, be sure to define them.

B. Research current literature to identify a recent project that has combined the processing capacity of numerous small computers to address a problem. Identify the operating system used to coordinate the processors for this project and discuss the obstacles overcome to make the system work. If your answer includes terms not used in this chapter, be sure to define them. Cite your sources.

Exercises

1. One friend claims that a dual-core system runs at twice the speed as a single-core system; another friend disagrees by saying that it runs twice as many programs at the same time; a third friend says it runs twice as much data at the same time. Explain who is correct and why.
2. Describe the relationship between a process and a thread in a multicore system.
3. Compare the processors' access to printers and other I/O devices for the master/slave and the symmetric multiprocessing configurations. Give a true-to-life example where the master/slave configuration might be preferred.
4. Compare the processors' access to main memory for the loosely coupled configuration and the symmetric multiprocessing configurations. Give a true-to-life example where the symmetric configuration might be preferred.

5. Describe the programmer's role when implementing explicit versus implicit parallelism.
6. What steps does a well-designed multiprocessing system follow when it detects that a processor is failing? What is the central goal of most multiprocessing systems?
7. Give an example from real life of busy waiting.
8. In the last chapter, we discussed deadlocks. Describe in your own words why mutual exclusion is necessary for multiprogramming systems.
9. Describe the purpose of a buffer and give an example from your own experience where its use clearly benefits system response.
10. Consider this formula:

$$G = (A + C^2) * (E - 1)^3 / D + B$$

    a. Show the order that a processor would follow to calculate G. To do so, break down the equation into the correct order of operations, with one calculation per step. Show the formula for each step, assigning new letters to calculations as necessary.
    b. Find the value of G: if A = 5, B = 10, C = 3, D = 8, and E = 5.
11. Consider this formula:

$$G = D + (A + C^2) * E / (D + B)^3$$

    a. Show the order that a processor would follow to calculate G. To do so, break down the equation into the correct order of operations with one calculation per step. Show the formula for each step, assigning new letters to calculations as necessary.
    b. Find the value of G: if A = 5, B = 10, C = 3, D = 8, and E = 5.
12. Rewrite each of the following arithmetic expressions to take advantage of concurrent processing and then code each. Use the terms COBEGIN and COEND to delimit the sections of concurrent code.
    a. $A + B * R * Z - N * M + C^2$
    b. $(X * (Y * Z * W * R) + M + N + P)$
    c. $((J + K * L * M * N) * I)$
13. Rewrite each of the following expressions for concurrent processing and then code each one. Use the terms COBEGIN and COEND to delimit the sections of concurrent code. Identify which expressions, if any, might *not* run faster in a concurrent processing environment.
    a. $H^2 * (O * (N + T))$
    b. $X * (Y * Z * W * R)$
    c. $M * T * R$

Advanced Exercises

14. Use the P and V semaphore operations to simulate the traffic flow at the intersection of two one-way streets. The following rules should be satisfied:
    - Only one car can cross at any given time.
    - A car should be allowed to cross the intersection only if there are no cars coming from the other street.
    - When cars are coming from both streets, they should take turns to prevent indefinite postponements in either street.
15. Explain the similarities and differences between the critical region and working set.
16. If a process terminates, will its threads also terminate, or will they continue to run? If all of its threads terminate, will the process also terminate, or will it continue to run? Explain your answers.
17. If a process is suspended (put into the "wait" state by an interrupt), will its threads also be suspended? Explain why the threads will or will not be suspended. Give an example to substantiate your answer.
18. Consider the following program segments for two different processes (P1, P2) executing concurrently and where B and A are not shared variables, but *x* starts at 0 and is a shared variable.

Processor #1	Processor #2
for(a = 1; a <= 3; a++)	for(b = 1; b <= 3; b++)
x = x + 1;	x = x + 1;

    If the processes P1 and P2 execute only once at any speed, what are the possible resulting values of *x*? Explain your answers.
19. Examine one of the programs you have written recently and indicate which operations could be executed concurrently. How long did it take you to do this? When might it be a good idea to write your programs in such a way that they can be run concurrently? Are you inclined to do so? Explain why or why not.
20. Consider the following segment taken from a C program:

```
for(j = 1; j <= 12; j++)
{
   printf(" nEnter an integer value:");
   scanf("%d", &x);
   if(x == 0)
   y(j)=0;
   if(x != 0)
   y(j)=10;
}
```

a. Recode it so it will run more efficiently in a single-processor system.
b. Given that a multiprocessing environment with four symmetrical processors is available, recode the segment as an efficient concurrent program that performs the same function as the original C program.
c. Given that all processors have identical capabilities, compare the execution times of the original C segment with the execution times of your segments for parts (a) and (b).

Programming Exercises

21. Edsger Dijkstra introduced the Sleeping Barber Problem (Dijkstra, 1965): A barbershop is divided into two rooms. The waiting room has *n* chairs, and the work room only has the barber chair. When the waiting room is empty, the barber goes to sleep in the barber chair. If a customer comes in and the barber is asleep, he knows it's his turn to get his hair cut. So he wakes up the barber and takes his turn in the barber chair. But if the waiting room is not empty, the customer must take a seat in the waiting room and wait his turn. Write a program that will coordinate the barber and his customers.
22. Suhas Patil introduced the Cigarette Smokers Problem (Patil, 1971): Three smokers and a supplier make up this system. Each smoker wants to roll a cigarette and smoke it immediately. However, to smoke a cigarette the smoker needs three ingredients—paper, tobacco, and a match. To the great discomfort of everyone involved, each smoker has only one of the ingredients: Smoker 1 has lots of paper, Smoker 2 has lots of tobacco, and Smoker 3 has the matches. And, of course, the rules of the group don't allow hoarding, swapping, or sharing. The supplier, who doesn't smoke, provides all three ingredients and has an infinite amount of all three items. But the supplier provides only two of them at a time—and only when no one is smoking. Here's how it works. The supplier randomly selects and places two different items on the table (where they are accessible to all three smokers), and the smoker with the remaining ingredient immediately takes them, rolls, and smokes a cigarette. Once done smoking, the first smoker signals the supplier, who then places another two randomly selected items on the table, and so on.

    Write a program that will synchronize the supplier with the smokers. Keep track of how many cigarettes each smoker consumes. Is this a fair supplier? Why or why not?