CHAPTER 5

Concurrency: Mutual Exclusion and Synchronization

Learning Objectives

After studying this chapter, you should be able to:

- Discuss basic concepts related to concurrency, such as race conditions, OS concerns, and mutual exclusion requirements.
- Understand hardware approaches to supporting mutual exclusion.
- Define and explain semaphores.
- Define and explain monitors.
- Explain the readers/writers problem.

The central themes of operating system design are all concerned with the management of processes and threads:

- **Multiprogramming:** The management of multiple processes within a uniprocessor system
- **Multiprocessing:** The management of multiple processes within a multiprocessor
- **Distributed processing:** The management of multiple processes executing on multiple, distributed computer systems. The recent proliferation of clusters is a prime example of this type of system.

Fundamental to all of these areas, and fundamental to OS design, is **concurrency**. Concurrency encompasses a host of design issues, including communication among processes, sharing of and competing for resources (such as memory, files, and I/O access), synchronization of the activities of multiple processes, and allocation of processor time to processes. We shall see that these issues arise not just in multiprocessing and distributed processing environments, but also in single-processor multiprogramming systems.

Concurrency arises in three different contexts:

1. **Multiple applications:** Multiprogramming was invented to allow processing time to be dynamically shared among a number of active applications.
2. **Structured applications:** As an extension of the principles of modular design and structured programming, some applications can be effectively programmed as a set of **concurrent processes**.
3. **Operating system structure:** The same structuring advantages apply to systems programs, and we have seen that operating systems are themselves often implemented as a set of processes or threads.

Because of the importance of this topic, four chapters and an appendix focus on concurrency-related issues. Chapters 5 and 6 will deal with concurrency in multiprogramming and multiprocessing systems. Chapters 16 and 18 will examine concurrency issues related to distributed processing.

This chapter begins with an introduction to the concept of concurrency and the implications of the execution of multiple concurrent processes.[1] We find that the basic requirement for support of concurrent processes is the ability to enforce mutual exclusion; that is, the ability to exclude all other processes from a course of action while one process is granted that ability. Section 5.2 covers various approaches to achieving mutual exclusion. All of these are software solutions that require the use of a technique known as busy waiting. Next, we will examine some hardware mechanisms that can support mutual exclusion. Then, we will look at solutions that do not involve busy waiting and that can be either supported by the OS or enforced by language compilers. We will examine three approaches: semaphores, monitors, and **message passing**.

Two classic problems in concurrency are used to illustrate the concepts and compare the approaches presented in this chapter. The producer/consumer problem will be introduced in Section 5.4 and used as a running example. The chapter closes with the readers/writers problem.

Our discussion of concurrency will continue in Chapter 6, and we defer a discussion of the concurrency mechanisms of our example systems until the end of that chapter. Appendix A covers additional topics on concurrency. Table 5.1 lists some key terms related to concurrency. A set of animations that illustrate concepts in this chapter is available at the Companion website for this book.

Table 5.1 Some Key Terms Related to Concurrency

Atomic operation	A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state. Atomicity guarantees isolation from concurrent processes.
Critical section	A section of code within a process that requires access to shared resources, and that must not be executed while another process is in a corresponding section of code.
Deadlock	A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.
Livelock	A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.
Mutual exclusion	The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.
Race condition	A situation in which multiple threads or processes read and write a shared data item, and the final result depends on the relative timing of their execution.
Starvation	A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

[1] For simplicity, we generally refer to the concurrent execution of *processes*. In fact, as we have seen in the preceding chapter, in some systems the fundamental unit of concurrency is a thread rather than a process.

5.1 MUTUAL EXCLUSION: SOFTWARE APPROACHES

Software approaches can be implemented for concurrent processes that execute on a single-processor or a multiprocessor machine with shared main memory. These approaches usually assume elementary mutual exclusion at the memory access level ([LAMP91], but see Problem 5.3). That is, simultaneous accesses (reading and/or writing) to the same location in main memory are serialized by some sort of memory arbiter, although the order of access granting is not specified ahead of time. Beyond this, no support in the hardware, operating system, or programming language is assumed.

Dekker's Algorithm

Dijkstra [DIJK65] reported an algorithm for mutual exclusion for two processes, designed by the Dutch mathematician Dekker. Following Dijkstra, we develop the solution in stages. This approach has the advantage of illustrating many of the common bugs encountered in developing concurrent programs.

First Attempt As mentioned earlier, any attempt at mutual exclusion must rely on some fundamental exclusion mechanism in the hardware. The most common of these is the constraint that only one access to a memory location can be made at a time. Using this constraint, we reserve a global memory location labeled `turn`. A process (P0 or P1) wishing to execute its critical section first examines the contents of `turn`. If the value of `turn` is equal to the number of the process, then the process may proceed to its critical section. Otherwise, it is forced to wait. Our waiting process repeatedly reads the value of `turn` until it is allowed to enter its critical section. This procedure is known as **busy waiting**, or **spin waiting**, because the thwarted process can do nothing productive until it gets permission to enter its critical section. Instead, it must linger and periodically check the variable; thus it consumes processor time (busy) while waiting for its chance.

After a process has gained access to its critical section, and after it has completed that section, it must update the value of `turn` to that of the other process.

In formal terms, there is a shared global variable:

```
int  turn = 0;
```

Figure 5.1a shows the program for the two processes. This solution guarantees the mutual exclusion property but has two drawbacks. First, processes must strictly alternate in their use of their critical section; therefore, the pace of execution is dictated by the slower of the two processes. If P0 uses its critical section only once per hour, but P1 would like to use its critical section at a rate of 1,000 times per hour, P1 is forced to adopt the pace of P0. A much more serious problem is that if one process fails, the other process is permanently blocked. This is true whether a process fails in its critical section or outside of it.

The foregoing construction is that of a **coroutine**. Coroutines are designed to be able to pass execution control back and forth between themselves (see Problem

```
   /* PROCESS 0 /*                  /* PROCESS 1 *
.                                  .
.                                  .
while (turn != 0)                  while (turn != 1)
   /* do nothing */ ;                 /* do nothing */;
/* critical section*/;              /* critical section*/;
turn = 1;                          turn = 0;
.                                  .
```

(a) First attempt

```
      /* PROCESS 0 *                /* PROCESS 1 *
   .                                  .
   .                                  .
   while (flag[1])                    while (flag[0])
      /* do nothing */;                  /* do nothing */;
   flag[0] = true;                    flag[1] = true;
   /*critical section*/;              /* critical section*/;
   flag[0] = false;                   flag[1] = false;
   .                                  .
```

(b) Second attempt

```
      /* PROCESS 0 *                /* PROCESS 1 *
   .                                  .
   .                                  .
   flag[0] = true;                    flag[1] = true;
   while (flag[1])                    while (flag[0])
      /* do nothing */;                  /* do nothing */;
   /* critical section*/;              /* critical section*/;
   flag[0] = false;                   flag[1] = false;
   .                                  .
```

(c) Third attempt

```
      /* PROCESS 0 *                /* PROCESS 1 *
   .                                  .
   .                                  .
   flag[0] = true;                    flag[1] = true;
   while (flag[1]) {                  while (flag[0]) {
      flag[0] = false;                   flag[1] = false;
      /*delay */;                        /*delay */;
      flag[0] = true;                    flag[1] = true;
   }                                  }
   /*critical section*/;              /* critical section*/;
   flag[0] = false;                   flag[1] = false;
   .
```

(d) Fourth attempt

VideoNote

Figure 5.1 Mutual Exclusion Attempts

5.5). While this is a useful structuring technique for a single process, it is inadequate to support concurrent processing.

SECOND ATTEMPT The flaw in the first attempt is that it stores the name of the process that may enter its critical section, when in fact we need state information about both processes. In effect, each process should have its own key to the critical section so that if one fails, the other can still access its critical section. To meet this requirement a Boolean vector `flag` is defined, with `flag[0]` corresponding to P0 and `flag[1]` corresponding to P1. Each process may examine the other's flag but may not alter it. When a process wishes to enter its critical section, it periodically checks the other's flag until that flag has the value `false`, indicating that the other process is not in its critical section. The checking process immediately sets its own flag to `true` and proceeds to its critical section. When it leaves its critical section, it sets its flag to `false`.

The shared global variable[2] now is

```
enum     boolean (false = 0; true = 1);
boolean  flag[2] = 0, 0
```

Figure 5.1b shows the algorithm. If one process fails outside the critical section, including the flag-setting code, then the other process is not blocked. In fact, the other process can enter its critical section as often as it likes, because the flag of the other process is always `false`. However, if a process fails inside its critical section or after setting its flag to `true` just before entering its critical section, then the other process is permanently blocked.

This solution is, if anything, worse than the first attempt because it does not even guarantee mutual exclusion. Consider the following sequence:

P0 executes the **while** statement and finds `flag[1]` set to `false`

Pl executes the **while** statement and finds `flag[0]` set to `false`

P0 sets `flag[0]` to `true` and enters its critical section

Pl sets `flag[1]` to `true` and enters its critical section

Because both processes are now in their critical sections, the program is incorrect. The problem is that the proposed solution is not independent of relative process execution speeds.

THIRD ATTEMPT Because a process can change its state after the other process has checked it but before the other process can enter its critical section, the second attempt failed. Perhaps we can fix this problem with a simple interchange of two statements, as shown in Figure 5.1c.

As before, if one process fails inside its critical section, including the flag-setting code controlling the critical section, then the other process is blocked, and if a process fails outside its critical section, then the other process is not blocked.

---

[2]The **enum** declaration is used here to declare a data type (`boolean`) and to assign its values.

Next, let us check that mutual exclusion is guaranteed, using the point of view of process P0. Once P0 has set `flag[0]` to `true`, Pl cannot enter its critical section until after P0 has entered and left its critical section. It could be that Pl is already in its critical section when P0 sets its flag. In that case, P0 will be blocked by the **while** statement until Pl has left its critical section. The same reasoning applies from the point of view of Pl.

This guarantees mutual exclusion, but creates yet another problem. If both processes set their flags to `true` before either has executed the **while** statement, then each will think that the other has entered its critical section, causing deadlock.

FOURTH ATTEMPT In the third attempt, a process sets its state without knowing the state of the other process. Deadlock occurs because each process can insist on its right to enter its critical section; there is no opportunity to back off from this position. We can try to fix this in a way that makes each process more deferential: Each process sets its flag to indicate its desire to enter its critical section, but is prepared to reset the flag to defer to the other process, as shown in Figure 5.1d.

This is close to a correct solution, but is still flawed. Mutual exclusion is still guaranteed, using similar reasoning to that followed in the discussion of the third attempt. However, consider the following sequence of events:

P0 sets `flag[0]` to `true`.
Pl sets `flag[1]` to `true`.
P0 checks `flag[1]`.
Pl checks `flag[0]`.
P0 sets `flag[0]` to `false`.
Pl sets `flag[1]` to `false`.
P0 sets `flag[0]` to `true`.
Pl sets `flag[1]` to `true`.

This sequence could be extended indefinitely, and neither process could enter its critical section. Strictly speaking, this is not deadlock, because any alteration in the relative speed of the two processes will break this cycle and allow one to enter the critical section. This condition is referred to as **livelock**. Recall that deadlock occurs when a set of processes wishes to enter their critical sections, but no process can succeed. With livelock, there are possible sequences of executions that succeed, but it is also possible to describe one or more execution sequences in which no process ever enters its critical section.

Although the scenario just described is not likely to be sustained for very long, it is nevertheless a possible scenario. Thus, we reject the fourth attempt.

A CORRECT SOLUTION We need to be able to observe the state of both processes, which is provided by the array variable `flag`. But, as the fourth attempt shows, this is not enough. We must impose an order on the activities of the two processes to avoid the problem of ”mutual courtesy” that we have just observed. The variable `turn`

from the first attempt can be used for this purpose; in this case the variable indicates which process has the right to insist on entering its critical region.

We can describe this solution, referred to as Dekker's algorithm, as follows. When P0 wants to enter its critical section, it sets its flag to `true`. It then checks the flag of Pl. If that is `false`, P0 may immediately enter its critical section. Otherwise, P0 consults `turn`. If P0 finds that `turn = 0`, then it knows that it is its turn to insist and periodically checks Pl's flag. Pl will at some point note that it is its turn to defer and set its to flag `false`, allowing P0 to proceed. After P0 has used its critical section, it sets its flag to `false` to free the critical section, and sets `turn` to l to transfer the right to insist to Pl.

```
boolean flag [2];
int turn;
void P0()
{
      while (true) {
            flag [0] = true;
            while (flag [1]) {
                  if (turn == 1)
                        flag [0] = false;
                        while (turn == 1) /* do nothing */;
                        flag [0] = true;
                  }
            }
            /* critical section */;
            turn = 1;
            flag [0] = false;
            /* remainder */;
      }
}
void P1( )
{
      while (true) {
            flag [1] = true;
            while (flag [0]) {
                  if (turn == 0) {
                        flag [1] = false;
                        while (turn == 0) /* do nothing */;
                        flag [1] = true;
                  }
            }
            /* critical section */;
            turn = 0;
            flag [1] = false;
            /* remainder */;
      }
}
void main ()
{
      flag [0] = false;
      flag [1] = false;
      turn = 1;
      parbegin  (P0, P1);
}
```


VideoNote

Figure 5.2 Dekker's Algorithm

Figure 5.2 provides a specification of Dekker's algorithm. The construct **parbegin** (Pl, P2, . . . , P*n*) means the following: suspend the execution of the main program; initiate concurrent execution of procedures Pl, P2, . . . , P*n*; when all of Pl, P2, . . . , P*n* have terminated, resume the main program. A verification of Dekker's algorithm is left as an exercise (see Problem 5.l).

Peterson's Algorithm

Dekker's algorithm solves the mutual exclusion problem, but with a rather complex program that is difficult to follow and whose correctness is tricky to prove. Peterson [PETE8l] has provided a simple, elegant solution. As before, the global array variable `flag` indicates the position of each process with respect to mutual exclusion, and the global variable `turn` resolves simultaneity conflicts. The algorithm is presented in Figure 5.3.

That mutual exclusion is preserved is easily shown. Consider process P0. Once it has set `flag[0]` to `true`, Pl cannot enter its critical section. If Pl already is in its critical section, then `flag[1] = true` and P0 is blocked from entering its critical section. On the other hand, mutual blocking is prevented. Suppose that P0 is blocked in its **while** loop. This means that `flag[1]` is true and `turn = 1`. P0 can

```
boolean flag [2];
int turn;
void P0()
{
      while (true) {
            flag [0] = true;
            turn = 1;
            while (flag [1] && turn == 1) /* do nothing */;
            /* critical section */;
            flag [0] = false;
            /* remainder */;
      }
}
 void P1()
{
      while (true) {
            flag [1] = true;
            turn = 0;
            while (flag [0] && turn == 0) /* do nothing */;
            /* critical section */;
            flag [1] = false;
            /* remainder */
      }
}
void main()
{
      flag [0] = false;
      flag [1] = false;
      parbegin  (P0, P1);
}
```

VideoNote **Figure 5.3 Peterson's Algorithm for Two Processes**

enter its critical section when either `flag[1]` becomes `false` or `turn` becomes 0. Now consider three exhaustive cases:

1. Pl has no interest in its critical section. This case is impossible, because it implies `flag[1] = false`.
2. Pl is waiting for its critical section. This case is also impossible, because if `turn = 1`, Pl is able to enter its critical section.
3. Pl is using its critical section repeatedly and therefore monopolizing access to it. This cannot happen, because Pl is obliged to give P0 an opportunity by setting `turn` to 0 before each attempt to enter its critical section.

Thus, we have a simple solution to the mutual exclusion problem for two processes. Furthermore, Peterson's algorithm is easily generalized to the case of processes [HOFR90].

5.2 PRINCIPLES OF CONCURRENCY

In a single-processor multiprogramming system, processes are interleaved in time to yield the appearance of simultaneous execution (see Figure 2.12a). Even though actual parallel processing is not achieved, and even though there is a certain amount of overhead involved in switching back and forth between processes, interleaved execution provides major benefits in processing efficiency and in program structuring. In a multiprocessor system, it is possible not only to interleave the execution of multiple processes, but also to overlap them (see Figure 2.12b).

At first glance, it may seem that interleaving and overlapping represent fundamentally different modes of execution and present different problems. In fact, both techniques can be viewed as examples of concurrent processing, and both present the same problems. In the case of a uniprocessor, the problems stem from a basic characteristic of multiprogramming systems: The relative speed of execution of processes cannot be predicted. It depends on the activities of other processes, the way in which the OS handles interrupts, and the scheduling policies of the OS. The following difficulties arise:

1. The sharing of global resources is fraught with peril. For example, if two processes both make use of the same global variable and both perform reads and writes on that variable, then the order in which the various reads and writes are executed is critical. An example of this problem is shown in the following subsection.
2. It is difficult for the OS to optimally manage the allocation of resources. For example, process A may request use of, and be granted control of, a particular I/O channel, then be suspended before using that channel. It may be undesirable for the OS simply to lock the channel and prevent its use by other processes; indeed this may lead to a deadlock condition, as will be described in Chapter 6.
3. It becomes very difficult to locate a programming error because results are typically not deterministic and reproducible (e.g., see [LEBL87, CARR89, SHEN02] for a discussion of this point).

All of the foregoing difficulties present themselves in a multiprocessor system as well, because here too the relative speed of execution of processes is unpredictable. A multiprocessor system must also deal with problems arising from the simultaneous execution of multiple processes. Fundamentally, however, the problems are the same as those for uniprocessor systems. This should become clear as the discussion proceeds.

A Simple Example

Consider the following procedure:

```
void echo()
{
  chin = getchar();
  chout = chin;
  putchar(chout);
}
```

This procedure shows the essential elements of a program that will provide a character `echo` procedure; input is obtained from a keyboard one keystroke at a time. Each input character is stored in variable `chin`. The character is then transferred to variable `chout` and sent to the display. Any program can call this procedure repeatedly to accept user input and display it on the user's screen.

Now consider that we have a single-processor multiprogramming system supporting a single user. The user can jump from one application to another, and each application uses the same keyboard for input and the same screen for output. Because each application needs to use the procedure `echo`, it makes sense for it to be a shared procedure that is loaded into a portion of memory global to all applications. Thus, only a single copy of the `echo` procedure is used, saving space.

The sharing of main memory among processes is useful to permit efficient and close interaction among processes. However, such sharing can lead to problems. Consider the following sequence:

1. Process P1 invokes the `echo` procedure and is interrupted immediately after `getchar` returns its value and stores it in `chin`. At this point, the most recently entered character, x, is stored in variable `chin`.
2. Process P2 is activated and invokes the `echo` procedure, which runs to conclusion, inputting and then displaying a single character, `y`, on the screen.
3. Process P1 is resumed. By this time, the value `x` has been overwritten in `chin` and therefore lost. Instead, `chin` contains `y`, which is transferred to `chout` and displayed.

Thus, the first character is lost and the second character is displayed twice. The essence of this problem is the shared global variable, `chin`. Multiple processes have access to this variable. If one process updates the global variable and is then interrupted, another process may alter the variable before the first process can use its value. Suppose, however, that we permit only one process at a time to be in that procedure. Then the foregoing sequence would result in the following:

1. Process P1 invokes the `echo` procedure and is interrupted immediately after the conclusion of the input function. At this point, the most recently entered character, `x`, is stored in variable `chin`.
2. Process P2 is activated and invokes the `echo` procedure. However, because P1 is still inside the `echo` procedure, although currently suspended, P2 is blocked from entering the procedure. Therefore, P2 is suspended awaiting the availability of the `echo` procedure.
3. At some later time, process P1 is resumed and completes execution of `echo`. The proper character, `x`, is displayed.
4. When P1 exits `echo`, this removes the block on P2. When P2 is later resumed, the `echo` procedure is successfully invoked.

This example shows that it is necessary to protect shared global variables (and other shared global resources) and the only way to do that is to control the code that accesses the variable. If we impose the discipline that only one process at a time may enter `echo`, and that once in `echo` the procedure must run to completion before it is available for another process, then the type of error just discussed will not occur. How that discipline may be imposed is a major topic of this chapter.

This problem was stated with the assumption that there was a single-processor, multiprogramming OS. The example demonstrates that the problems of concurrency occur even when there is a single processor. In a multiprocessor system, the same problems of protected shared resources arise, and the same solution works. First, suppose there is no mechanism for controlling access to the shared global variable:

1. Processes P1 and P2 are both executing, each on a separate processor. Both processes invoke the `echo` procedure.
2. The following events occur; events on the same line take place in parallel:

Process P1	**Process P2**
•	•
`chin = getchar();`	•
•	`chin = getchar();`
`chout = chin;`	`chout = chin;`
`putchar(chout);`	•
•	`putchar(chout);`
•	•

The result is that the character input to P1 is lost before being displayed, and the character input to P2 is displayed by both P1 and P2. Again, let us add the capability of enforcing the discipline that only one process at a time may be in `echo`. Then the following sequence occurs:

1. Processes P1 and P2 are both executing, each on a separate processor. P1 invokes the `echo` procedure.
2. While P1 is inside the `echo` procedure, P2 invokes `echo`. Because P1 is still inside the `echo` procedure (whether P1 is suspended or executing), P2 is

blocked from entering the procedure. Therefore, P2 is suspended awaiting the availability of the `echo` procedure.

3. At a later time, process P1 completes execution of `echo`, exits that procedure, and continues executing. Immediately upon the exit of P1 from `echo`, P2 is resumed and begins executing `echo`.

In the case of a uniprocessor system, the reason we have a problem is that an interrupt can stop instruction execution anywhere in a process. In the case of a multiprocessor system, we have that same condition and, in addition, a problem can be caused because two processes may be executing simultaneously and both trying to access the same global variable. However, the solution to both types of problem is the same: control access to the shared resource.

Race Condition

A race condition occurs when multiple processes or threads read and write data items so that the final result depends on the order of execution of instructions in the multiple processes. Let us consider two simple examples.

As a first example, suppose two processes, P1 and P2, share the global variable `a`. At some point in its execution, P1 updates `a` to the value 1, and at some point in its execution, P2 updates `a` to the value 2. Thus, the two tasks are in a race to write variable `a`. In this example, the "loser" of the race (the process that updates last) determines the final value of `a`.

For our second example, consider two processes, P3 and P4, that share global variables `b` and `c`, with initial values `b = 1` and `c = 2`. At some point in its execution, P3 executes the assignment `b = b + c`, and at some point in its execution, P4 executes the assignment `c = b + c`. Note the two processes update different variables. However, the final values of the two variables depend on the order in which the two processes execute these two assignments. If P3 executes its assignment statement first, then the final values are `b = 3` and `c = 5`. If P4 executes its assignment statement first, then the final values are `b = 4` and `c = 3`.

Appendix A includes a discussion of race conditions using semaphores as an example.

Operating System Concerns

What design and management issues are raised by the existence of concurrency? We can list the following concerns:

1. The OS must be able to keep track of the various processes. This is done with the use of process control blocks and was described in Chapter 4.
2. The OS must allocate and deallocate various resources for each active process. At times, multiple processes want access to the same resource. These resources include
   - **Processor time:** This is the scheduling function, to be discussed in Part Four.
   - **Memory:** Most operating systems use a virtual memory scheme. The topic will be addressed in Part Three.

- **Files:** To be discussed in Chapter 12
- **I/O devices:** To be discussed in Chapter 11

3. The OS must protect the data and physical resources of each process against unintended interference by other processes. This involves techniques that relate to memory, files, and I/O devices. A general treatment of protection found in Part Seven.
4. The functioning of a process, and the output it produces, must be independent of the speed at which its execution is carried out relative to the speed of other concurrent processes. This is the subject of this chapter.

To understand how the issue of speed independence can be addressed, we need to look at the ways in which processes can interact.

Process Interaction

We can classify the ways in which processes interact on the basis of the degree to which they are aware of each other's existence. Table 5.2 lists three possible degrees of awareness and the consequences of each:

- **Processes unaware of each other:** These are independent processes that are not intended to work together. The best example of this situation is the multiprogramming of multiple independent processes. These can either be batch jobs or interactive sessions or a mixture. Although the processes are not working together, the OS needs to be concerned about **competition** for resources. For example, two independent applications may both want to access the same disk or file or printer. The OS must regulate these accesses.

Table 5.2 Process Interaction

Degree of Awareness	Relationship	Influence that One Process Has on the Other	Potential Control Problems
Processes unaware of each other	Competition	• Results of one process independent of the action of others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation • Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Deadlock (consumable resource) • Starvation

- **Processes indirectly aware of each other:** These are processes that are not necessarily aware of each other by their respective process IDs but that share access to some object, such as an I/O buffer. Such processes exhibit **cooperation** in sharing the common object.
- **Processes directly aware of each other:** These are processes that are able to communicate with each other by process ID and that are designed to work jointly on some activity. Again, such processes exhibit **cooperation**.

Conditions will not always be as clear-cut as suggested in Table 5.2. Rather, several processes may exhibit aspects of both competition and cooperation. Nevertheless, it is productive to examine each of the three items in the preceding list separately and determine their implications for the OS.

Competition among Processes for Resources Concurrent processes come into conflict with each other when they are competing for the use of the same resource. In its pure form, we can describe the situation as follows. Two or more processes need to access a resource during the course of their execution. Each process is unaware of the existence of other processes, and each is to be unaffected by the execution of the other processes. It follows from this each process should leave the state of any resource that it uses unaffected. Examples of resources include I/O devices, memory, processor time, and the clock.

There is no exchange of information between the competing processes. However, the execution of one process may affect the behavior of competing processes. In particular, if two processes both wish access to a single resource, then one process will be allocated that resource by the OS, and the other will have to wait. Therefore, the process that is denied access will be slowed down. In an extreme case, the blocked process may never get access to the resource, and hence will never terminate successfully.

In the case of competing processes three control problems must be faced. First is the need for **mutual exclusion**. Suppose two or more processes require access to a single nonsharable resource, such as a printer. During the course of execution, each process will be sending commands to the I/O device, receiving status information, sending data, and/or receiving data. We will refer to such a resource as a **critical resource**, and the portion of the program that uses it as a **critical section** of the program. It is important that only one program at a time be allowed in its critical section. We cannot simply rely on the OS to understand and enforce this restriction because the detailed requirements may not be obvious. In the case of the printer, for example, we want any individual process to have control of the printer while it prints an entire file. Otherwise, lines from competing processes will be interleaved.

The enforcement of mutual exclusion creates two additional control problems. One is that of **deadlock**. For example, consider two processes, P1 and P2, and two resources, R1 and R2. Suppose that each process needs access to both resources to perform part of its function. Then it is possible to have the following situation: the OS assigns R1 to P2, and R2 to P1. Each process is waiting for one of the two resources. Neither will release the resource that it already owns until it has acquired the other resource and performed the function requiring both resources. The two processes are deadlocked.

```
/* PROCESS 1 * void  P1
{
  while  (true) {
   /* preceding code */;
   entercritical (Ra);
   /* critical section */;
   exitcritical (Ra);
   /* following code */;
  }
}
```

```
/* PROCESS 2 * void  P2
{
  while  (true) {
   /* preceding code */;
   entercritical (Ra);
   /* critical section */;
   exitcritical (Ra);
   /* following code */;
  }
}
```

• • •

```
/* PROCESS n * void  Pn
{
  while  (true) {
   /* preceding code */;
   entercritical (Ra);
   /* critical section */;
   exitcritical (Ra);
   /* following code */;
  }
}
```

VideoNote

Figure 5.4 Illustration of Mutual Exclusion

A final control problem is **starvation**. Suppose three processes (P1, P2, P3) each require periodic access to resource R. Consider the situation in which P1 is in possession of the resource, and both P2 and P3 are delayed, waiting for that resource. When P1 exits its critical section, either P2 or P3 should be allowed access to R. Assume the OS grants access to P3, and P1 again requires access before P3 completes its critical section. If the OS grants access to P1 after P3 has finished, and subsequently alternately grants access to P1 and P3, then P2 may indefinitely be denied access to the resource, even though there is no deadlock situation.

Control of competition inevitably involves the OS because the OS allocates resources. In addition, the processes themselves will need to be able to express the requirement for mutual exclusion in some fashion, such as locking a resource prior to its use. Any solution will involve some support from the OS, such as the provision of the locking facility. Figure 5.4 illustrates the mutual exclusion mechanism in abstract terms. There are *n* processes to be executed concurrently. Each process includes (1) a critical section that operates on some resource Ra, and (2) additional code preceding and following the critical section that does not involve access to Ra. Because all processes access the same resource Ra, it is desired that only one process at a time be in its critical section. To enforce mutual exclusion, two functions are provided: `entercritical` and `exitcritical`. Each function takes as an argument the name of the resource that is the subject of competition. Any process that attempts to enter its critical section while another process is in its critical section, for the same resource, is made to wait.

It remains to examine specific mechanisms for providing the functions `entercritical` and `exitcritical`. For the moment, we defer this issue while we consider the other cases of process interaction.

Cooperation among Processes by Sharing The case of cooperation by sharing covers processes that interact with other processes without being explicitly aware of them. For example, multiple processes may have access to shared variables or to shared files or databases. Processes may use and update the shared data without reference to other processes, but know that other processes may have access to the same data. Thus the processes must cooperate to ensure that the data they share are properly managed. The control mechanisms must ensure the integrity of the shared data.

Because data are held on resources (devices, memory), the control problems of mutual exclusion, deadlock, and starvation are again present. The only difference is that data items may be accessed in two different modes, reading and writing, and only writing operations must be mutually exclusive.

However, over and above these problems, a new requirement is introduced: that of data coherence. As a simple example, consider a bookkeeping application in which various data items may be updated. Suppose two items of data a and b are to be maintained in the relationship $a = b$. That is, any program that updates one value must also update the other to maintain the relationship. Now consider the following two processes:

P1:
```
a = a + 1;
b = b + 1;
```
P2:
```
b = 2 * b;
a = 2 * a;
```

If the state is initially consistent, each process taken separately will leave the shared data in a consistent state. Now consider the following concurrent execution sequence, in which the two processes respect mutual exclusion on each individual data item (a and b):

```
a = a + 1;
b = 2 * b;
b = b + 1;
a = 2 * a;
```

At the end of this execution sequence, the condition $a = b$ no longer holds. For example, if we start with $a = b = 1$, at the end of this execution sequence we have $a = 4$ and $b = 3$. The problem can be avoided by declaring the entire sequence in each process to be a critical section.

Thus, we see that the concept of critical section is important in the case of cooperation by sharing. The same abstract functions of `entercritical` and `exitcritical` discussed earlier (see Figure 5.4) can be used here. In this case, the argument for the functions could be a variable, a file, or any other shared object. Furthermore, if critical sections are used to provide data integrity, then there may be no specific resource or variable that can be identified as an argument. In that case, we can think of the argument as being an identifier that is shared among concurrent processes to identify critical sections that must be mutually exclusive.

Cooperation among Processes by Communication In the first two cases that we have discussed, each process has its own isolated environment that does not include the other processes. The interactions among processes are indirect. In both cases, there is a sharing. In the case of competition, they are sharing resources without being aware of the other processes. In the second case, they are sharing values, and although each process is not explicitly aware of the other processes, it is aware of the need to maintain data integrity. When processes cooperate by communication, however, the various processes participate in a common effort that links all of the processes. The communication provides a way to synchronize, or coordinate, the various activities.

Typically, communication can be characterized as consisting of messages of some sort. Primitives for sending and receiving messages may be provided as part of the programming language or provided by the OS kernel.

Because nothing is shared between processes in the act of passing messages, mutual exclusion is not a control requirement for this sort of cooperation. However, the problems of deadlock and starvation are still present. As an example of deadlock, two processes may be blocked, each waiting for a communication from the other. As an example of starvation, consider three processes, P1, P2, and P3, that exhibit the following behavior. P1 is repeatedly attempting to communicate with either P2 or P3, and P2 and P3 are both attempting to communicate with P1. A sequence could arise in which P1 and P2 exchange information repeatedly, while P3 is blocked waiting for a communication from P1. There is no deadlock, because P1 remains active, but P3 is starved.

Requirements for Mutual Exclusion

Any facility or capability that is to provide support for mutual exclusion should meet the following requirements:

1. Mutual exclusion must be enforced: Only one process at a time is allowed into its critical section, among all processes that have critical sections for the same resource or shared object.
2. A process that halts in its noncritical section must do so without interfering with other processes.
3. It must not be possible for a process requiring access to a critical section to be delayed indefinitely: no deadlock or starvation.
4. When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay.
5. No assumptions are made about relative process speeds or number of processors.
6. A process remains inside its critical section for a finite time only.

There are a number of ways in which the requirements for mutual exclusion can be satisfied. One approach is to leave the responsibility with the processes that wish to execute concurrently. Processes, whether they are system programs or application programs, would be required to coordinate with one another to enforce mutual exclusion, with no support from the programming language or the OS. We can refer to these as software approaches. Although this approach is prone to high processing overhead and bugs, it is nevertheless useful to examine such approaches to gain a better understanding of the complexity of concurrent processing. This topic was covered in the preceding section. A second approach involves the use of special-purpose machine instructions. These have the advantage of reducing overhead but nevertheless will be shown to be unattractive as a general-purpose solution; they will be covered in Section 5.3. A third approach is to provide some level of support within the OS or a programming language. Three of the most important such approaches will be examined in Sections 5.4 through 5.6.

5.3 MUTUAL EXCLUSION: HARDWARE SUPPORT

In this section, we look at several interesting hardware approaches to mutual exclusion.

Interrupt Disabling

In a uniprocessor system, concurrent processes cannot have overlapped execution; they can only be interleaved. Furthermore, a process will continue to run until it invokes an OS service or until it is interrupted. Therefore, to guarantee mutual exclusion, it is sufficient to prevent a process from being interrupted. This capability can be provided in the form of primitives defined by the OS kernel for disabling and enabling interrupts. A process can then enforce mutual exclusion in the following way (compare to Figure 5.4):

```
while (true) {
    /* disable interrupts */;
    /* critical section */;
    /* enable interrupts */;
    /* remainder */;
}
```

Because the critical section cannot be interrupted, mutual exclusion is guaranteed. The price of this approach, however, is high. The efficiency of execution could be noticeably degraded because the processor is limited in its ability to interleave processes. Another problem is that this approach will not work in a multiprocessor architecture. When the computer includes more than one processor, it is possible (and typical) for more than one process to be executing at a time. In this case, disabled interrupts do not guarantee mutual exclusion.

Special Machine Instructions

In a multiprocessor configuration, several processors share access to a common main memory. In this case, there is not a master/slave relationship; rather the processors behave independently in a peer relationship. There is no interrupt mechanism between processors on which mutual exclusion can be based.

At the hardware level, as was mentioned, access to a memory location excludes any other access to that same location. With this as a foundation, processor designers have proposed several machine instructions that carry out two actions atomically,[3] such as reading and writing or reading and testing, of a single memory location with one instruction fetch cycle. During execution of the instruction, access to the memory location is blocked for any other instruction referencing that location.

In this section, we look at two of the most commonly implemented instructions. Others are described in [RAYN86] and [STON93].

---

[3]The term *atomic* means the instruction is treated as a single step that cannot be interrupted.

COMPARE&SWAP INSTRUCTION The `compare&swap` instruction, also called a compare and exchange instruction, can be defined as follows [HERL90]:

```
int compare_and_swap (int *word, int testval, int newval)
{
    int oldval;
    oldval = *word
    if (oldval == testval) *word = newval;
    return oldval;
}
```

This version of the instruction checks a memory location (`*word`) against a test value (`testval`). If the memory location's current value is `testval`, it is replaced with `newval`; otherwise, it is left unchanged. The old memory value is always returned; thus, the memory location has been updated if the returned value is the same as the test value. This atomic instruction therefore has two parts: A **compare** is made between a memory value and a test value; if the values are the same, a **swap** occurs. The entire `compare&swap` function is carried out atomically—that is, it is not subject to interruption.

Another version of this instruction returns a Boolean value: true if the swap occurred; false otherwise. Some version of this instruction is available on nearly all processor families (x86, IA64, sparc, IBM z series, etc.), and most operating systems use this instruction for support of concurrency.

Figure 5.5a shows a mutual exclusion protocol based on the use of this instruction.[4] A shared variable `bolt` is initialized to 0. The only process that may enter its

```
/* program  mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int  i)
{
   while (true)  {
     while (compare_and_swap(bolt, 0, 1) == 1)
          /* do nothing */;
     /* critical section */;
     bolt = 0;
     /* remainder */;
   }
}
void main() {
   bolt = 0;
   parbegin (P(1), P(2), ... ,P(n));
}
```

(a) Compare and swap instruction

```
/* program mutualexclusion */
int const n = /* number of processes */;
int bolt;
void P(int  i)
{
   while (true)
      int  keyi = 1;
      do  exchange (&keyi, &bolt)
      while  (keyi != 0);
      /* critical section */;
      bolt = 0;
      /* remainder */;
   }
}
void main() {
   bolt = 0;
   parbegin  (P(1), P(2), ..., P(n));
}
```

(b) Exchange instruction

VideoNote **Figure 5.5 Hardware Support for Mutual Exclusion**

[4] The construct **parbegin** `(P1, P2, . . . , Pn)` means the following: suspend the execution of the main program; initiate concurrent execution of procedures P1, P2, . . . , P*n*; when all of P1, P2, . . . , P*n* have terminated, resume the main program.

critical section is one that finds `bolt` equal to 0. All other processes attempting to enter their critical section go into a busy waiting mode. The term **busy waiting,** or **spin waiting,** refers to a technique in which a process can do nothing until it gets permission to enter its critical section, but continues to execute an instruction or set of instructions that tests the appropriate variable to gain entrance. When a process leaves its critical section, it resets *bolt* to 0; at this point one and only one of the waiting processes is granted access to its critical section. The choice of process depends on which process happens to execute the `compare&swap` instruction next.

EXCHANGE INSTRUCTION The exchange instruction can be defined as follows:

```
void exchange (int *register,  int *memory)
{
    int temp;
    temp = *memory;
    *memory = *register;
    *register = temp;
}
```

The instruction exchanges the contents of a register with that of a memory location. Both the Intel IA-32 architecture (Pentium) and the IA-64 architecture (Itanium) contain an XCHG instruction.

Figure 5.5b shows a mutual exclusion protocol based on the use of an exchange instruction. A shared variable *bolt* is initialized to 0. Each process uses a local variable *key* that is initialized to 1. The only process that may enter its critical section is one that finds *bolt* equal to 0. It excludes all other processes from the critical section by setting *bolt* to 1. When a process leaves its critical section, it resets *bolt* to 0, allowing another process to gain access to its critical section.

Note the following expression always holds because of the way in which the variables are initialized and because of the nature of the exchange algorithm:

$$bolt + \sum_i key_i = n$$

If $bolt = 0$, then no process is in its critical section. If $bolt = 1$, then exactly one process is in its critical section, namely the process whose *key* value equals 0.

PROPERTIES OF THE MACHINE-INSTRUCTION APPROACH The use of a special machine instruction to enforce mutual exclusion has a number of advantages:

- It is applicable to any number of processes on either a single processor or multiple processors sharing main memory.
- It is simple and therefore easy to verify.
- It can be used to support multiple critical sections; each critical section can be defined by its own variable.

However, there are some serious disadvantages:

- **Busy waiting is employed:** Thus, while a process is waiting for access to a critical section, it continues to consume processor time.

- **Starvation is possible:** When a process leaves a critical section and more than one process is waiting, the selection of a waiting process is arbitrary. Thus, some process could indefinitely be denied access.
- **Deadlock is possible:** Consider the following scenario on a single-processor system. Process P1 executes the special instruction (e.g., `compare&swap`, `exchange`) and enters its critical section. P1 is then interrupted to give the processor to P2, which has higher priority. If P2 now attempts to use the same resource as P1, it will be denied access because of the mutual exclusion mechanism. Thus, it will go into a busy waiting loop. However, P1 will never be dispatched because it is of lower priority than another ready process, P2.

Because of the drawbacks of both the software and hardware solutions, we need to look for other mechanisms.

5.4 SEMAPHORES

We now turn to OS and programming language mechanisms that are used to provide concurrency. Table 5.3 summarizes mechanisms in common use. We begin, in this section, with semaphores. The next two sections will discuss monitors and message passing. The other mechanisms in Table 5.3 will be discussed when treating specific OS examples, in Chapters 6 and 13.

Table 5.3 Common Concurrency Mechanisms

Semaphore	An integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment. The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. Also known as a **counting semaphore** or a **general semaphore**.
Binary semaphore	A semaphore that takes on only the values 0 and 1.
Mutex	Similar to a binary semaphore. A key difference between the two is that the process that locks the mutex (sets the value to 0) must be the one to unlock it (sets the value to 1).
Condition variable	A data type that is used to block a process or thread until a particular condition is true.
Monitor	A programming language construct that encapsulates variables, access procedures, and initialization code within an abstract data type. The monitor's variable may only be accessed via its access procedures and only one process may be actively accessing the monitor at any one time. The access procedures are *critical sections*. A monitor may have a queue of processes that are waiting to access it.
Event flags	A memory word used as a synchronization mechanism. Application code may associate a different event with each bit in a flag. A thread can wait for either a single event or a combination of events by checking one or multiple bits in the corresponding flag. The thread is blocked until all of the required bits are set (AND) or until at least one of the bits is set (OR).
Mailboxes/messages	A means for two processes to exchange information and that may be used for synchronization.
Spinlocks	Mutual exclusion mechanism in which a process executes in an infinite loop waiting for the value of a lock variable to indicate availability.

The first major advance in dealing with the problems of concurrent processes came in 1965 with Dijkstra's treatise [DIJK65]. Dijkstra was concerned with the design of an OS as a collection of cooperating sequential processes, and with the development of efficient and reliable mechanisms for supporting cooperation. These mechanisms can just as readily be used by user processes if the processor and OS make the mechanisms available.

The fundamental principle is this: Two or more processes can cooperate by means of simple signals, such that a process can be forced to stop at a specified place until it has received a specific signal. Any complex coordination requirement can be satisfied by the appropriate structure of signals. For signaling, special variables called semaphores are used. To transmit a signal via semaphore `s`, a process executes the primitive `semSignal (s)`. To receive a signal via semaphore `s`, a process executes the primitive `semWait (s)`; if the corresponding signal has not yet been transmitted, the process is suspended until the transmission takes place.[5]

To achieve the desired effect, we can view the semaphore as a variable that has an integer value upon which only three operations are defined:

1. A semaphore may be initialized to a nonnegative integer value.
2. The `semWait` operation decrements the semaphore value. If the value becomes negative, then the process executing the `semWait` is blocked. Otherwise, the process continues execution.
3. The `semSignal` operation increments the semaphore value. If the resulting value is less than or equal to zero, then a process blocked by a `semWait` operation, if any, is unblocked.

Other than these three operations, there is no way to inspect or manipulate semaphores.

We explain these operations as follows. To begin, the semaphore has a zero or positive value. If the value is positive, that value equals the number of processes that can issue a wait and immediately continue to execute. If the value is zero, either by initialization or because a number of processes equal to the initial semaphore value have issued a wait, the next process to issue a wait is blocked, and the semaphore value goes negative. Each subsequent wait drives the semaphore value further into minus territory. The negative value equals the number of processes waiting to be unblocked. Each signal unblocks one of the waiting processes when the semaphore value is negative.

[Subject] points out three interesting consequences of the semaphore definition:

1. In general, there is no way to know before a process decrements a semaphore whether it will block or not.

[5] In Dijkstra's original paper and in much of the literature, the letter P is used for `semWait` and the letter V for `semSignal`; these are the initials of the Dutch words for test (*proberen*) and increment (*verhogen*). In some of the literature, the terms `wait` and `signal` are used. This book uses `semWait` and `semSignal` for clarity, and to avoid confusion with similar wait and signal operations in monitors, discussed subsequently.

```
struct semaphore {
        int count;
        queueType queue;
};
void semWait(semaphore s)
{
        s.count--;
        if (s.count < 0) {
          /* place this process in s.queue */;
          /* block this process */;
        }
}
void semSignal(semaphore s)
{
        s.count++;
        if (s.count<= 0) {
          /* remove a process P from s.queue */;
          /* place process P on ready list */;
        }
}
```


VideoNote **Figure 5.6 A Definition of Semaphore Primitives**

2. After a process increments a semaphore and another process gets woken up, both processes continue running concurrently. There is no way to know which process, if either, will continue immediately on a uniprocessor system.
3. When you signal a semaphore, you don't necessarily know whether another process is waiting, so the number of unblocked processes may be zero or one.

Figure 5.6 suggests a more formal definition of the primitives for semaphores. The `semWait` and `semSignal` primitives are assumed to be atomic. A more restricted version, known as the **binary semaphore**, is defined in Figure 5.7. A binary semaphore may only take on the values 0 and 1, and can be defined by the following three operations:

1. A binary semaphore may be initialized to 0 or 1.
2. The `semWaitB` operation checks the semaphore value. If the value is zero, then the process executing the `semWaitB` is blocked. If the value is one, then the value is changed to zero and the process continues execution.
3. The `semSignalB` operation checks to see if any processes are blocked on this semaphore (semaphore value equals 0). If so, then a process blocked by a `semWaitB` operation is unblocked. If no processes are blocked, then the value of the semaphore is set to one.

In principle, it should be easier to implement the binary semaphore, and it can be shown that it has the same expressive power as the general semaphore (see Problem 5.19). To contrast the two types of semaphores, the nonbinary semaphore is often referred to as either a **counting semaphore** or a **general semaphore**.

```
struct binary_semaphore {
       enum {zero, one} value;
       queueType queue;
};
 void semWaitB(binary_semaphore s)
{
       if (s.value == one)
           s.value = zero;
       else {
                   /* place this process in s.queue */;
                   /* block this process */;
       }
}
 void semSignalB(semaphore s)
{
       if (s.queue is empty())
           s.value = one;
       else {
                   /* remove a process P from s.queue */;
                   /* place process P on ready list */;
       }
}
```

VideoNote **Figure 5.7 A Definition of Binary Semaphore Primitives**

A concept related to the binary semaphore is the **mutual exclusion lock (mutex)**. A mutex is a programming flag used to grab and release an object. When data are acquired that cannot be shared, or processing is started that cannot be performed simultaneously elsewhere in the system, the mutex is set to lock (typically zero), which blocks other attempts to use it. The mutex is set to unlock when the data are no longer needed or the routine is finished. A key difference between the a mutex and a binary semaphore is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1). In contrast, it is possible for one process to lock a binary semaphore and for another to unlock it.[6]

For both counting semaphores and binary semaphores, a queue is used to hold processes waiting on the semaphore. The question arises of the order in which processes are removed from such a queue. The fairest removal policy is first-in-first-out (FIFO): The process that has been blocked the longest is released from the queue first; a semaphore whose definition includes this policy is called a **strong semaphore**. A semaphore that does not specify the order in which processes are removed from the queue is a **weak semaphore**. Figure 5.8 is an example of the operation of a strong semaphore. Here processes A, B, and C depend on a result from process D. Initially (1), A is running; B, C, and D are ready; and the semaphore count is 1, indicating that one of D's results is available. When A issues a `semWait` instruction on semaphore *s*, the semaphore decrements to 0, and A can continue to execute; subsequently

[6]In some of the literature, and in some textbooks, no distinction is made between a mutex and a binary semaphore. However, in practice, a number of operating systems, such as Linux, Windows, and Solaris, offer a mutex facility which conforms to the definition in this book.

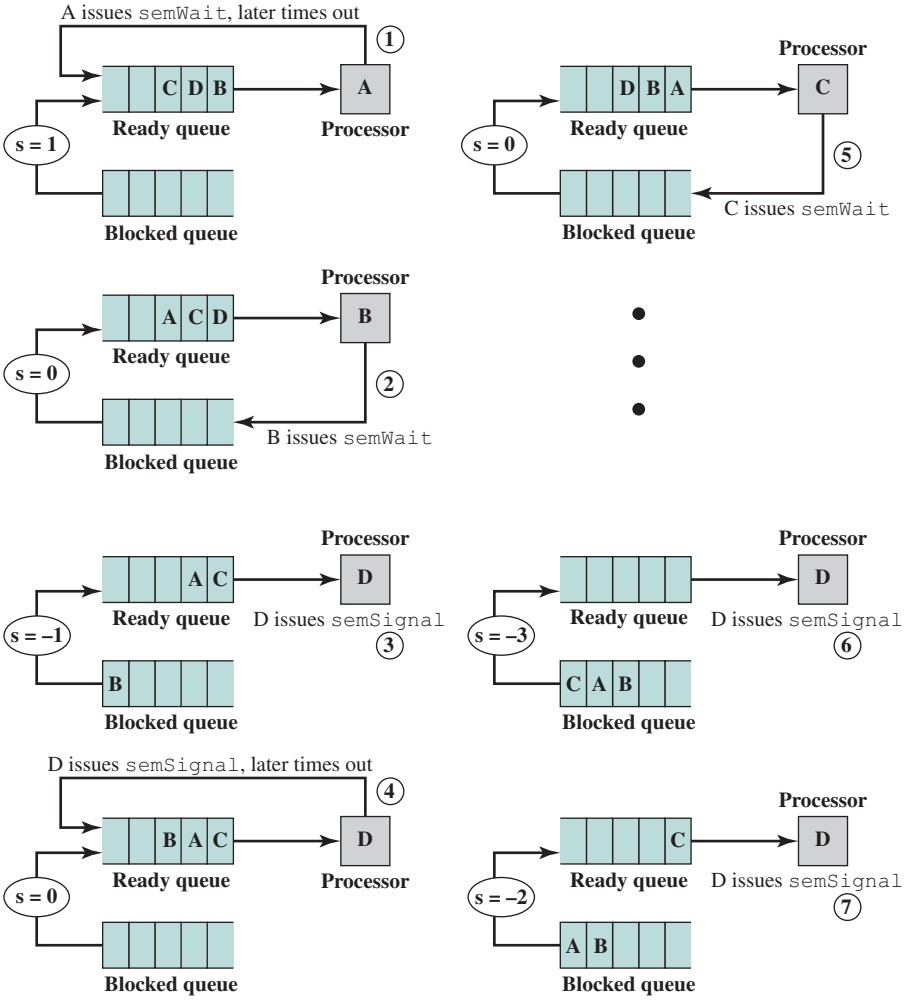


Figure 5.8 **Example of Semaphore Mechanism**

it rejoins the ready queue. Then B runs (2), eventually issues a `semWait` instruction, and is blocked, allowing D to run (3). When D completes a new result, it issues a `semSignal` instruction, which allows B to move to the ready queue (4). D rejoins the ready queue and C begins to run (5) but is blocked when it issues a `semWait` instruction. Similarly, A and B run and are blocked on the semaphore, allowing D to resume execution (6). When D has a result, it issues a `semSignal`, which transfers C to the ready queue. Later cycles of D will release A and B from the Blocked state.

For the mutual exclusion algorithm discussed in the next subsection and illustrated in Figure 5.9, strong semaphores guarantee freedom from starvation, while weak semaphores do not. We will assume strong semaphores because they are more convenient, and because this is the form of semaphore typically provided by operating systems.

```
/*  program mutualexclusion */
const int n = /* number of processes */;
semaphore s = 1;
void P(int  i)
{
     while (true) {
          semWait(s);
          /* critical section */;
          semSignal(s);
          /* remainder */;
     }
}
void main()
{
     parbegin (P(1), P(2), . . . , P(n));
}
```

VideoNote **Figure 5.9 Mutual Exclusion Using Semaphores**

Mutual Exclusion

Figure 5.9 shows a straightforward solution to the mutual exclusion problem using a semaphore *s* (compare to Figure 5.4). Consider *n* processes, identified in the array *P*(*i*), all of which need access to the same resource. Each process has a critical section used to access the resource. In each process, a `semWait (s)` is executed just before its critical section. If the value of `s` becomes negative, the process is blocked. If the value is 1, then it is decremented to 0 and the process immediately enters its critical section; because `s` is no longer positive, no other process will be able to enter its critical section.

The semaphore is initialized to 1. Thus, the first process that executes a `semWait` will be able to enter the critical section immediately, setting the value of `s` to 0. Any other process attempting to enter the critical section will find it busy and will be blocked, setting the value of `s` to −1. Any number of processes may attempt entry; each such unsuccessful attempt results in a further decrement of the value of `s`. When the process that initially entered its critical section departs, `s` is incremented and one of the blocked processes (if any) is removed from the queue of blocked processes associated with the semaphore and put in a Ready state. When it is next scheduled by the OS, it may enter the critical section.

Figure 5.10, based on one in [BACO03], shows a possible sequence for three processes using the mutual exclusion discipline of Figure 5.9. In this example three processes (A, B, C) access a shared resource protected by the semaphore *lock*. Process A executes `semWait (`*lock*`)`; because the semaphore has a value of 1 at the time of the `semWait` operation, A can immediately enter its critical section and the semaphore takes on the value 0. While A is in its critical section, both B and C perform a `semWait` operation and are blocked pending the availability of the semaphore. When A exits its critical section and performs `semSignal (`*lock*`)`, B, which was the first process in the queue, can now enter its critical section.

The program of Figure 5.9 can equally well handle a requirement that more than one process be allowed in its critical section at a time. This requirement is met

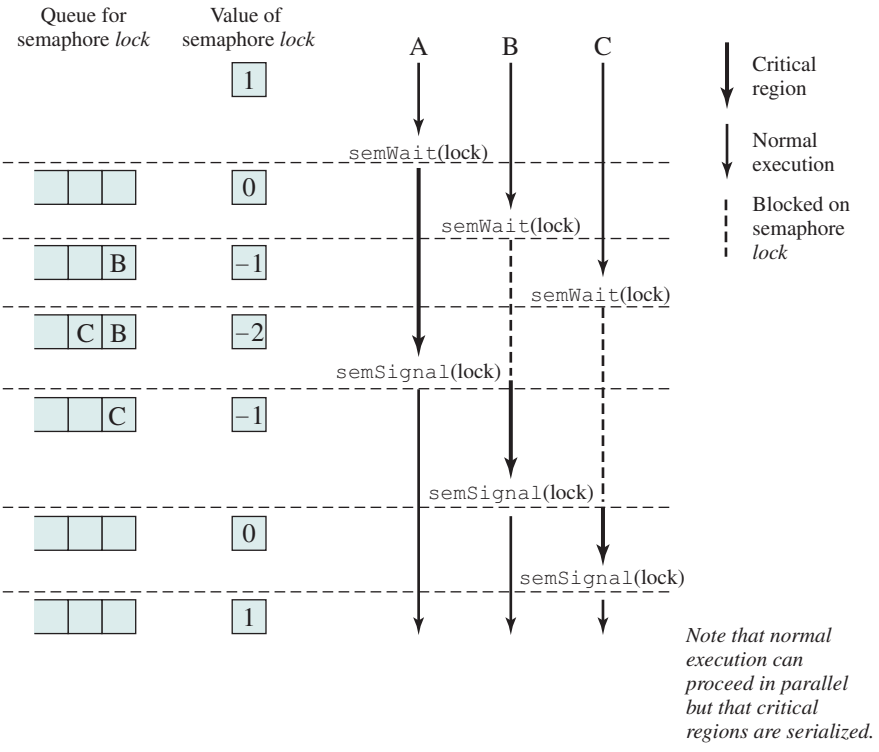


Figure 5.10 **Processes Accessing Shared Data Protected by a Semaphore**

simply by initializing the semaphore to the specified value. Thus, at any time, the value of *s.count* can be interpreted as follows:

- *s.count* $\geq$ 0: *s.count* is the number of processes that can execute `semWait (s)` without suspension (if no `semSignal (s)` is executed in the meantime). Such situations will allow semaphores to support synchronization as well as mutual exclusion.
- *s.count* $<$ 0: The magnitude of *s.count* is the number of processes suspended in *s.queue*.

The Producer/Consumer Problem

We now examine one of the most common problems faced in concurrent processing: the producer/consumer problem. The general statement is this: There are one or more producers generating some type of data (records, characters) and placing these in a buffer. There is a single consumer that is taking items out of the buffer one at a time. The system is to be constrained to prevent the overlap of buffer operations. That is, only one agent (producer or consumer) may access the buffer at any one time. The problem is to make sure that the producer won't try to add data into the buffer if it's full, and that the consumer won't try to remove data from an empty buffer. We will

look at a number of solutions to this problem to illustrate both the power and the pitfalls of semaphores.

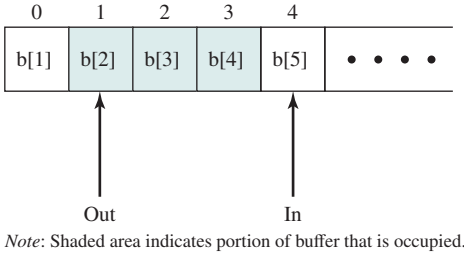
To begin, let us assume that the buffer is infinite and consists of a linear array of elements. In abstract terms, we can define the producer and consumer functions as follows:

```
producer:                          consumer:
while (true) {                     while (true) {
    /* produce item v */;               while (in <= out)
    b[in] = v;                              /* do nothing */;
    in++;                              w = b[out];
}                                      out++;
                                       /* consume item w */;
                                   }
```

Figure 5.11 illustrates the structure of buffer `b`. The producer can generate items and store them in the buffer at its own pace. Each time, an index (*in*) into the buffer is incremented. The consumer proceeds in a similar fashion but must make sure that it does not attempt to read from an empty buffer. Hence, the consumer makes sure that the producer has advanced beyond it (*in* > *out*) before proceeding.

Let us try to implement this system using binary semaphores. Figure 5.12 is a first attempt. Rather than deal with the indices *in* and *out*, we can simply keep track of the number of items in the buffer, using the integer variable *n* (= *in* − *out*). The semaphore `s` is used to enforce mutual exclusion; the semaphore `delay` is used to force the consumer to `semWait` if the buffer is empty.

This solution seems rather straightforward. The producer is free to add to the buffer at any time. It performs `semWaitB (s)` before appending and `semSignalB (s)` afterward to prevent the consumer (or any other producer) from accessing the buffer during the append operation. Also, while in the critical section, the producer increments the value of *n*. If $n = 1$, then the buffer was empty just prior to this append, so the producer performs `semSignalB (delay)` to alert the consumer of this fact. The consumer begins by waiting for the first item to be produced, using `semWaitB (delay)`. It then takes an item and decrements *n* in its critical section. If the producer is able to stay ahead of the consumer (a common situation), then the



Note: Shaded area indicates portion of buffer that is occupied.

Figure 5.11 Infinite Buffer for the Producer/Consumer Problem

```
/*  program producerconsumer */
      int n;
      binary_semaphore s = 1, delay = 0;
      void producer()
      {
           while (true) {
                 produce();
                 semWaitB(s);
                 append();
                 n++;
                 if (n==1) semSignalB(delay);
                 semSignalB(s);
           }
      }
      void consumer()
      {
           semWaitB(delay);
           while (true) {
                 semWaitB(s);
                 take();
                 n--;
                 semSignalB(s);
                 consume();
                 if (n==0) semWaitB(delay);
           }
      }
      void main()
      {
           n = 0;
           parbegin (producer, consumer);
      }
```

VideoNote **Figure 5.12 An Incorrect Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores**

consumer will rarely block on the semaphore `delay` because *n* will usually be positive. Hence, both producer and consumer run smoothly.

There is, however, a flaw in this program. When the consumer has exhausted the buffer, it needs to reset the delay semaphore so it will be forced to wait until the producer has placed more items in the buffer. This is the purpose of the statement: **`if`** `n == 0 semWaitB (delay)`. Consider the scenario outlined in Table 5.4. In line 14, the consumer fails to execute the `semWaitB` operation. The consumer did indeed exhaust the buffer and set *n* to 0 (line 8), but the producer has incremented *n* before the consumer can test it in line 14. The result is a `semSignalB` not matched by a prior `semWaitB`. The value of −1 for *n* in line 20 means the consumer has consumed an item from the buffer that does not exist. It would not do simply to move the conditional statement inside the critical section of the consumer, because this could lead to deadlock (e.g., after line 8 of Table 5.4).

A fix for the problem is to introduce an auxiliary variable that can be set in the consumer's critical section for use later on. This is shown in Figure 5.13. A careful trace of the logic should convince you that deadlock can no longer occur.

A somewhat cleaner solution can be obtained if general semaphores (also called counting semaphores) are used, as shown in Figure 5.14. The variable *n* is now

Table 5.4 Possible Scenario for the Program of Figure 5.12

	Producer	Consumer	s	*n*	Delay
1			1	0	0
2	semWaitB(s)		0	0	0
3	n++		0	1	0
4	**if** (n==1) (semSignalB(delay))		0	1	1
5	semSignalB(s)		1	1	1
6		semWaitB(delay)	1	1	0
7		semWaitB(s)	0	1	0
8		n--	0	0	0
9		semSignalB(s)	1	0	0
10	semWaitB(s)		0	0	0
11	n++		0	1	0
12	**if** (n==1) (semSignalB(delay))		0	1	1
13	semSignalB(s)		1	1	1
14		**if** (n==0) (semWaitB(delay))	1	1	1
15		semWaitB(s)	0	1	1
16		n--	0	0	1
17		semSignalB(s)	1	0	1
18		**if** (n==0) (semWaitB(delay))	1	0	0
19		semWaitB(s)	0	0	0
20		n--	0	−1	0
21		semSignalB(s)	1	−1	0

Note: White areas represent the critical section controlled by semaphore s.

a semaphore. Its value still is equal to the number of items in the buffer. Suppose now that in transcribing this program, a mistake is made and the operations `semSignal (s)` and `semSignal (n)` are interchanged. This would require that the `semSignal (n)` operation be performed in the producer's critical section without interruption by the consumer or another producer. Would this affect the program? No, because the consumer must wait on both semaphores before proceeding in any case.

Now suppose the `semWait (n)` and `semWait (s)` operations are accidentally reversed. This produces a serious, indeed a fatal, flaw. If the consumer ever enters its critical section when the buffer is empty ($n.count = 0$), then no producer can ever append to the buffer and the system is deadlocked. This is a good example of the subtlety of semaphores and the difficulty of producing correct designs.

Finally, let us add a new and realistic restriction to the producer/consumer problem: namely, that the buffer is finite. The buffer is treated as a circular storage

```
/* program producerconsumer */
      int n;
      binary_semaphore s = 1, delay = 0;
      void producer()
      {
           while (true) {
                 produce();
                 semWaitB(s);
                 append();
                 n++;
                 if (n==1) semSignalB(delay);
                 semSignalB(s);
           }
      }
      void  consumer()
      {
           int  m; /* a local variable */
           semWaitB(delay);
           while  (true) {
                 semWaitB(s);
                 take();
                 n--;
                 m = n;
                 semSignalB(s);
                 consume();
                 if  (m==0) semWaitB(delay);
           }
      }
      void  main()
      {
           n = 0;
           parbegin  (producer, consumer);
      }
```

VideoNote **Figure 5.13 A Correct Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores**

(see Figure 5.15), and pointer values must be expressed modulo the size of the buffer. The following relationships hold:

Block on:	Unblock on:
Producer: insert in full buffer	Consumer: item inserted
Consumer: remove from empty buffer	Producer: item removed

The producer and consumer functions can be expressed as follows (variable *in* and *out* are initialized to 0 and *n* is the size of the buffer):

```
producer:
while (true) {
    /* produce item v */
    while ((in + 1) % n == out)
       /* do nothing */;
    b[in] = v;
    in = (in + 1)  %  n;
}
```

```
consumer:
while (true) {
    while (in == out)
       /* do nothing */;
    w = b[out];
    out = (out + 1) % n;
    /* consume item w */;
}
```

```
/* program producerconsumer */
      semaphore n = 0, s = 1;
      void producer()
      {
           while (true) {
                 produce();
                 semWait(s);
                 append();
                 semSignal(s);
                 semSignal(n);
           }
      }
      void consumer()
      {
           while (true) {
                 semWait(n);
                 semWait(s);
                 take();
                 semSignal(s);
                 consume();
           }
      }
      void main()
      {
                parbegin  (producer, consumer);
      }
```

VideoNote **Figure 5.14 A Solution to the Infinite-Buffer Producer/Consumer Problem Using Semaphores**

Figure 5.16 shows a solution using general semaphores. The semaphore *e* has been added to keep track of the number of empty spaces.

Another instructive example in the use of semaphores is the barbershop problem described in Appendix A. Appendix A also includes additional examples of the problem of race conditions when using semaphores.

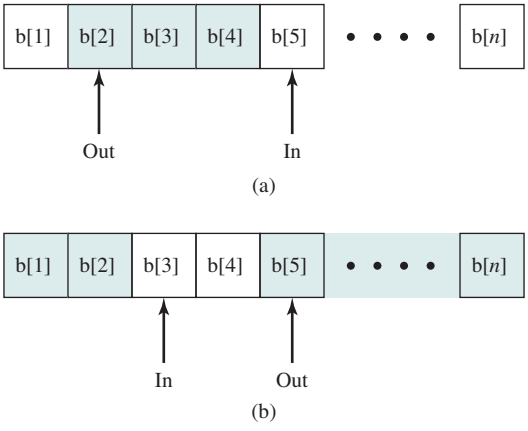


Figure 5.15 Finite Circular Buffer for the Producer/Consumer Problem

```
/* program boundedbuffer */
      const int sizeofbuffer = /* buffer size */;
      semaphore s = 1, n = 0, e = sizeofbuffer;
      void producer()
      {
           while (true) {
                 produce();
                 semWait(e);
                 semWait(s);
                 append();
                 semSignal(s);
                 semSignal(n);
           }
      }
      void consumer()
      {
           while  (true) {
                 semWait(n);
                 semWait(s);
                 take();
                 semSignal(s);
                 semSignal(e);
                 consume();
           }
      }
      void main()
      {
           parbegin  (producer, consumer);
      }
```

Figure 5.16 A Solution to the Bounded-Buffer Producer/Consumer Problem Using Semaphores

Implementation of Semaphores

As was mentioned earlier, it is imperative that the `semWait` and `semSignal` operations be implemented as atomic primitives. One obvious way is to implement them in hardware or firmware. Failing this, a variety of schemes have been suggested. The essence of the problem is one of mutual exclusion: Only one process at a time may manipulate a semaphore with either a `semWait` or `semSignal` operation. Thus, any of the software schemes, such as Dekker's algorithm or Peterson's algorithm (see Section 5.1), could be used; this would entail a substantial processing overhead.

Another alternative is to use one of the hardware-supported schemes for mutual exclusion. For example, Figure 5.17 shows the use of a `compare&swap` instruction. In this implementation, the semaphore is again a structure, as in Figure 5.6, but now includes a new integer component, *s.flag*. Admittedly, this involves a form of busy waiting. However, the `semWait` and `semSignal` operations are relatively short, so the amount of busy waiting involved should be minor.

For a single-processor system, it is possible to inhibit interrupts for the duration of a `semWait` or `semSignal` operation, as suggested in Figure 5.17b. Once again, the relatively short duration of these operations means that this approach is reasonable.

```
semWait(s)
{
   while (compare_and_swap(s.flag, 0 , 1) == 1)
      /* do nothing */;
   s.count--;
   if (s.count < 0) {
      /* place this process in s.queue*/;
      /* block this process (must also set
s.flag to 0) */;
   }
   s.flag = 0;
}
semSignal(s)
{
   while (compare_and_swap(s.flag, 0 , 1) == 1)
      /* do nothing */;
   s.count++;
   if  (s.count<= 0) {
      /* remove a process P from s.queue */;
      /* place process P on ready list */;
   }
   s.flag = 0;
}
```

(a) Compare and Swap Instruction

```
semWait(s)
{
   inhibit interrupts;
   s.count--;
   if (s.count < 0) {
      /* place this process in s.queue */;
      /* block this process and allow inter-
rupts*/;
   }
   else
      allow interrupts;
}

semSignal(s)
{
   inhibit interrupts;
   s.count++;
   if (s.count<= 0) {
      /* remove a process P from s.queue */;
      /* place process P on ready list */;
   }
   allow interrupts;
}
```

(b) Interrupts

VideoNote **Figure 5.17 Two Possible Implementations of Semaphores**

5.5 MONITORS

Semaphores provide a primitive yet powerful and flexible tool for enforcing mutual exclusion and for coordinating processes. However, as Figure 5.12 suggests, it may be difficult to produce a correct program using semaphores. The difficulty is that `semWait` and `semSignal` operations may be scattered throughout a program, and it is not easy to see the overall effect of these operations on the semaphores they affect.

The monitor is a programming language construct that provides equivalent functionality to that of semaphores and that is easier to control. The concept was first formally defined in [HOAR74]. The monitor construct has been implemented in a number of programming languages, including Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, and Java. It has also been implemented as a program library. This allows programmers to put a monitor lock on any object. In particular, for something like a linked list, you may want to lock all linked lists with one lock, or have one lock for each list, or have one lock for each element of each list.

We begin with a look at Hoare's version, and then examine a refinement.

Monitor with Signal

A monitor is a software module consisting of one or more procedures, an initialization sequence, and local data. The chief characteristics of a monitor are the following:

1. The local data variables are accessible only by the monitor's procedures and not by any external procedure.

2. A process enters the monitor by invoking one of its procedures.
3. Only one process may be executing in the monitor at a time; any other processes that have invoked the monitor are blocked, waiting for the monitor to become available.

The first two characteristics are reminiscent of those for objects in object-oriented software. Indeed, an object-oriented OS or programming language can readily implement a monitor as an object with special characteristics.

By enforcing the discipline of one process at a time, the monitor is able to provide a mutual exclusion facility. The data variables in the monitor can be accessed by only one process at a time. Thus, a shared data structure can be protected by placing it in a monitor. If the data in a monitor represent some resource, then the monitor provides a mutual exclusion facility for accessing the resource.

To be useful for concurrent processing, the monitor must include synchronization tools. For example, suppose a process invokes the monitor and, while in the monitor, must be blocked until some condition is satisfied. A facility is needed by which the process is not only blocked, but releases the monitor so some other process may enter it. Later, when the condition is satisfied and the monitor is again available, the process needs to be resumed and allowed to reenter the monitor at the point of its suspension.

A monitor supports synchronization by the use of **condition variables** that are contained within the monitor and accessible only within the monitor. Condition variables are a special data type in monitors, which are operated on by two functions:

- `cwait(c)`: Suspend execution of the calling process on condition *c*. The monitor is now available for use by another process.
- `csignal(c)`: Resume execution of some process blocked after a `cwait` on the same condition. If there are several such processes, choose one of them; if there is no such process, do nothing.

Note that monitor *wait* and *signal* operations are different from those for the semaphore. If a process in a monitor signals and no task is waiting on the condition variable, the signal is lost.

Figure 5.18 illustrates the structure of a monitor. Although a process can enter the monitor by invoking any of its procedures, we can think of the monitor as having a single entry point that is guarded so only one process may be in the monitor at a time. Other processes that attempt to enter the monitor join a queue of processes blocked waiting for monitor availability. Once a process is in the monitor, it may temporarily block itself on condition `x` by issuing `cwait(x)`; it is then placed in a queue of processes waiting to reenter the monitor when the condition changes, and resume execution at the point in its program following the `cwait(x)` call.

If a process that is executing in the monitor detects a change in the condition variable `x`, it issues `csignal(x)`, which alerts the corresponding condition queue that the condition has changed.

As an example of the use of a monitor, let us return to the bounded-buffer producer/consumer problem. Figure 5.19 shows a solution using a monitor. The monitor module, `boundedbuffer`, controls the buffer used to store and retrieve characters. The monitor includes two condition variables (declared with the

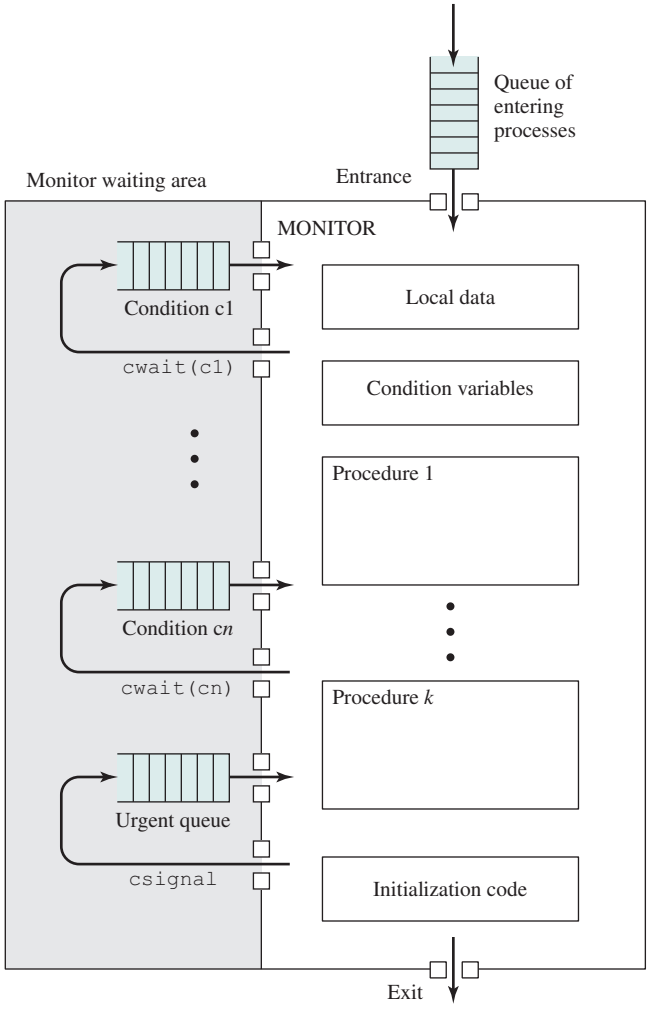


Figure 5.18 Structure of a Monitor

construct **cond**): *notfull* is true when there is room to add at least one character to the buffer, and *notempty* is true when there is at least one character in the buffer.

A producer can add characters to the buffer only by means of the procedure append inside the monitor; the producer does not have direct access to *buffer*. The procedure first checks the condition *notfull* to determine if there is space available in the buffer. If not, the process executing the monitor is blocked on that condition. Some other process (producer or consumer) may now enter the monitor. Later, when the buffer is no longer full, the blocked process may be removed from the queue, reactivated, and resume processing. After placing a character in the buffer, the process signals the *notempty* condition. A similar description can be made of the consumer function.

This example points out the division of responsibility with monitors compared to semaphores. In the case of monitors, the monitor construct itself enforces mutual

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];                                          /* space for N items */
int nextin, nextout;                                        /* buffer pointers */
int count;                                             /* number of items in buffer */
cond notfull, notempty;              /* condition variables for synchronization */
void append (char x)

{
      if (count == N) cwait(notfull);          /* buffer is full; avoid overflow */
      buffer[nextin] = x;
      nextin = (nextin + 1) % N;
      count++;
      /* one more item in buffer */
      csignal (notempty);                           /*resume any waiting consumer */
}
void take (char x)
{
      if (count == 0) cwait(notempty);       /* buffer is empty; avoid underflow */
      x = buffer[nextout];
      nextout = (nextout + 1) % N);
      count--;                                          /* one fewer item in buffer */
      csignal (notfull);                           /* resume any waiting producer */
}
{                                                                  /* monitor body */
      nextin = 0; nextout = 0; count = 0;              /* buffer initially empty */
}
```

```
void producer()
{
      char x;
      while (true) {
      produce(x);
      append(x);
      }
}
void consumer()
{
      char x;
      while (true) {
      take(x);
      consume(x);
      }
}
void main()
{
      parbegin (producer, consumer);
}
```

VideoNote **Figure 5.19 A Solution to the Bounded-Buffer Producer/Consumer Problem Using a Monitor**

exclusion: It is not possible for both a producer and a consumer to simultaneously access the buffer. However, the programmer must place the appropriate `cwait` and `csignal` primitives inside the monitor to prevent processes from depositing items in a full buffer or removing them from an empty one. In the case of semaphores, both mutual exclusion and synchronization are the responsibility of the programmer.

Note in Figure 5.19, a process exits the monitor immediately after executing the `csignal` function. If the `csignal` does not occur at the end of the procedure, then, in Hoare's proposal, the process issuing the signal is blocked to make the monitor available and placed in a queue until the monitor is free. One possibility at this point would be to place the blocked process in the entrance queue, so it would have to compete for access with other processes that had not yet entered the monitor. However, because a process blocked on a `csignal` function has already partially performed its task in the monitor, it makes sense to give this process precedence over newly entering processes by setting up a separate urgent queue (see Figure 5.18). One language that uses monitors, Concurrent Pascal, requires that `csignal` only appear as the last operation executed by a monitor procedure.

If there are no processes waiting on condition *x*, then the execution of `csignal(x)` has no effect.

As with semaphores, it is possible to make mistakes in the synchronization function of monitors. For example, if either of the `csignal` functions in the `bounded_buffer` monitor are omitted, then processes entering the corresponding condition queue are permanently hung up. The advantage that monitors have over semaphores is that all of the synchronization functions are confined to the monitor. Therefore, it is easier to verify that the synchronization has been done correctly and to detect bugs. Furthermore, once a monitor is correctly programmed, access to the protected resource is correct for access from all processes. In contrast, with semaphores, resource access is correct only if all of the processes that access the resource are programmed correctly.

Alternate Model of Monitors with Notify and Broadcast

Hoare's definition of monitors [HOAR74] requires that if there is at least one process in a condition queue, a process from that queue runs immediately when another process issues a `csignal` for that condition. Thus, the process issuing the `csignal` must either immediately exit the monitor or be blocked on the monitor.

There are two drawbacks to this approach:

1. If the process issuing the `csignal` has not finished with the monitor, then two additional process switches are required: one to block this process, and another to resume it when the monitor becomes available.
2. Process scheduling associated with a signal must be perfectly reliable. When a `csignal` is issued, a process from the corresponding condition queue must be activated immediately, and the scheduler must ensure that no other process enters the monitor before activation. Otherwise, the condition under which the process was activated could change. For example, in Figure 5.19, when a `csignal(notempty)` is issued, a process from the `notempty` queue must be activated before a new consumer enters the monitor. Another example: A producer process may append a character to an empty buffer then fail before signaling; any processes in the `notempty` queue would be permanently hung up.

Lampson and Redell developed a different definition of monitors for the language Mesa [LAMP80]. Their approach overcomes the problems just listed and supports several useful extensions. The Mesa monitor structure is also used in the Modula-3 systems programming language [NELS91]. In Mesa, the `csignal` primitive

```
void append (char x)
{
      while (count == N) cwait(notfull);       /* buffer is full; avoid overflow */
      buffer[nextin] = x;
      nextin = (nextin + 1) % N;
      count++;                                        /* one more item in buffer */
      cnotify(notempty);                           /* notify any waiting consumer */
}
void take (char x)
{
      while (count == 0) cwait(notempty);    /* buffer is empty; avoid underflow */
      x = buffer[nextout];
      nextout = (nextout + 1) % N);
      count--;                                          /* one fewer item in buffer */
      cnotify(notfull);                            /* notify any waiting producer */
}
```


VideoNote

Figure 5.20 Bounded-Buffer Monitor Code for Mesa Monitor

is replaced by `cnotify`, with the following interpretation: When a process executing in a monitor executes `cnotify(x)`, it causes the *x* condition queue to be notified, but the signaling process continues to execute. The result of the notification is that the process at the head of the condition queue will be resumed at some convenient future time when the monitor is available. However, because there is no guarantee that some other process will not enter the monitor before the waiting process, the waiting process must recheck the condition. For example, the procedures in the `boundedbuffer` monitor would now have the code of Figure 5.20.

The **if** statements are replaced by **while** loops. Thus, this arrangement results in at least one extra evaluation of the condition variable. In return, however, there are no extra process switches, and no constraints on when the waiting process must run after a `cnotify`.

One useful refinement that can be associated with the `cnotify` primitive is a watchdog timer associated with each condition primitive. A process that has been waiting for the maximum timeout interval will be placed in a ready state regardless of whether the condition has been notified. When activated, the process checks the condition and continues if the condition is satisfied. The timeout prevents the indefinite starvation of a process in the event that some other process fails before signaling a condition.

With the rule that a process is notified rather than forcibly reactivated, it is possible to add a `cbroadcast` primitive to the repertoire. The broadcast causes all processes waiting on a condition to be placed in a ready state. This is convenient in situations where a process does not know how many other processes should be reactivated. For example, in the producer/consumer program, suppose both the append and the `take` functions can apply to variable-length blocks of characters. In that case, if a producer adds a block of characters to the buffer, it need not know how many characters each waiting consumer is prepared to consume. It simply issues a `cbroadcast`, and all waiting processes are alerted to try again.

In addition, a broadcast can be used when a process would have difficulty figuring out precisely which other process to reactivate. A good example is a memory

manager. The manager has j bytes free; a process frees up an additional k bytes, but it does not know which waiting process can proceed with a total of $k + j$ bytes. Hence it uses broadcast, and all processes check for themselves if there is enough memory free.

An advantage of Lampson/Redell monitors over Hoare monitors is that the Lampson/Redell approach is less prone to error. In the Lampson/Redell approach, because each procedure checks the monitor variable after being signaled, with the use of the **while** construct, a process can signal or broadcast incorrectly without causing an error in the signaled program. The signaled program will check the relevant variable and, if the desired condition is not met, continue to wait.

Another advantage of the Lampson/Redell monitor is that it lends itself to a more modular approach to program construction. For example, consider the implementation of a buffer allocator. There are two levels of conditions to be satisfied for cooperating sequential processes:

1. Consistent data structures. Thus, the monitor enforces mutual exclusion and completes an input or output operation before allowing another operation on the buffer.
2. Level 1, plus enough memory for this process to complete its allocation request.

In the Hoare monitor, each signal conveys the level 1 condition but also carries the implicit message, "I have freed enough bytes for your particular allocate call to work now." Thus, the signal implicitly carries the level 2 condition. If the programmer later changes the definition of the level 2 condition, it will be necessary to reprogram all signaling processes. If the programmer changes the assumptions made by any particular waiting process (i.e., waiting for a slightly different level 2 invariant), it may be necessary to reprogram all signaling processes. This is unmodular and likely to cause synchronization errors (e.g., wake up by mistake) when the code is modified. The programmer has to remember to modify all procedures in the monitor every time a small change is made to the level 2 condition. With a Lampson/Redell monitor, a broadcast ensures the level 1 condition and carries a hint that level 2 might hold; each process should check the level 2 condition itself. If a change is made in the level 2 condition in either a waiter or a signaler, there is no possibility of erroneous wakeup because each procedure checks its own level 2 condition. Therefore, the level 2 condition can be hidden within each procedure. With the Hoare monitor, the level 2 condition must be carried from the waiter into the code of every signaling process, which violates data abstraction and interprocedural modularity principles.

5.6 MESSAGE PASSING

When processes interact with one another, two fundamental requirements must be satisfied: synchronization and communication. Processes need to be synchronized to enforce mutual exclusion; cooperating processes may need to exchange information. One approach to providing both of these functions is message passing. Message passing has the further advantage that it lends itself to implementation in distributed systems as well as in shared-memory multiprocessor and uniprocessor systems.

Table 5.5 Design Characteristics of Message Systems for Interprocess Communication and Synchronization

Synchronization	**Format**
Send	Content
blocking	Length
nonblocking	fixed
Receive	variable
blocking	
nonblocking	**Queueing Discipline**
test for arrival	FIFO
	Priority
Addressing	
Direct	
send	
receive	
explicit	
implicit	
Indirect	
static	
dynamic	
ownership	

Message-passing systems come in many forms. In this section, we will provide a general introduction that discusses features typically found in such systems. The actual function of message passing is normally provided in the form of a pair of primitives:

```
send (destination, message)
receive (source, message)
```

This is the minimum set of operations needed for processes to engage in message passing. A process sends information in the form of a *message* to another process designated by a *destination*. A process receives information by executing the `receive` primitive, indicating the *source* and the *message*.

A number of design issues relating to message-passing systems are listed in Table 5.5, and examined in the remainder of this section.

Synchronization

The communication of a message between two processes implies some level of synchronization between the two: The receiver cannot receive a message until it has been sent by another process. In addition, we need to specify what happens to a process after it issues a `send` or `receive` primitive.

Consider the `send` primitive first. When a send primitive is executed in a process, there are two possibilities: Either the sending process is blocked until the

message is received, or it is not. Similarly, when a process issues a `receive` primitive, there are two possibilities:

1. If a message has previously been sent, the message is received and execution continues.
2. If there is no waiting message, then either (a) the process is blocked until a message arrives, or (b) the process continues to execute, abandoning the attempt to receive.

Thus, both the sender and receiver can be blocking or nonblocking. Three combinations are common, although any particular system will usually have only one or two combinations implemented:

1. **Blocking send, blocking receive:** Both the sender and receiver are blocked until the message is delivered; this is sometimes referred to as a *rendezvous*. This combination allows for tight synchronization between processes.
2. **Nonblocking send, blocking receive:** Although the sender may continue on, the receiver is blocked until the requested message arrives. This is probably the most useful combination. It allows a process to send one or more messages to a variety of destinations as quickly as possible. A process that must receive a message before it can do useful work needs to be blocked until such a message arrives. An example is a server process that exists to provide a service or resource to other processes.
3. **Nonblocking send, nonblocking receive:** Neither party is required to wait.

The nonblocking `send` is more natural for many concurrent programming tasks. For example, if it is used to request an output operation such as printing, it allows the requesting process to issue the request in the form of a message, then carry on. One potential danger of the nonblocking `send` is that an error could lead to a situation in which a process repeatedly generates messages. Because there is no blocking to discipline the process, these messages could consume system resources, including processor time and buffer space, to the detriment of other processes and the OS. Also, the nonblocking `send` places the burden on the programmer to determine that a message has been received: Processes must employ reply messages to acknowledge receipt of a message.

For the `receive` primitive, the blocking version appears to be more natural for many concurrent programming tasks. Generally, a process that requests a message will need the expected information before proceeding. However, if a message is lost, which can happen in a distributed system, or if a process fails before it sends an anticipated message, a receiving process could be blocked indefinitely. This problem can be solved by the use of the nonblocking `receive`. However, the danger of this approach is that if a message is sent after a process has already executed a matching `receive`, the message will be lost. Other possible approaches are to allow a process to test whether a message is waiting before issuing a `receive` and allow a process to specify more than one source in a `receive` primitive. The latter approach is useful if a process is waiting for messages from more than one source, and can proceed if any of these messages arrive.

Addressing

Clearly, it is necessary to have a way of specifying in the `send` primitive which process is to receive the message. Similarly, most implementations allow a receiving process to indicate the source of a message to be received.

The various schemes for specifying processes in `send` and `receive` primitives fall into two categories: direct addressing and indirect addressing. With **direct addressing**, the send primitive includes a specific identifier of the destination process. The `receive` primitive can be handled in one of two ways. One possibility is to require that the process explicitly designate a sending process. Thus, the process must know ahead of time from which process a message is expected. This will often be effective for cooperating concurrent processes. In other cases, however, it is impossible to specify the anticipated source process. An example is a printer-server process, which will accept a print request message from any other process. For such applications, a more effective approach is the use of implicit addressing. In this case, the *source* parameter of the `receive` primitive possesses a value returned when the receive operation has been performed.

The other general approach is **indirect addressing**. In this case, messages are not sent directly from sender to receiver, but rather are sent to a shared data structure consisting of queues that can temporarily hold messages. Such queues are generally referred to as *mailboxes*. Thus, for two processes to communicate, one process sends a message to the appropriate mailbox, and the other process picks up the message from the mailbox.

A strength of the use of indirect addressing is that, by decoupling the sender and receiver, it allows for greater flexibility in the use of messages. The relationship between senders and receivers can be one-to-one, many-to-one, one-to-many, or many-to-many (see Figure 5.21). A **one-to-one** relationship allows a private communications link to be set up between two processes. This insulates their interaction from erroneous interference from other processes. A **many-to-one** relationship is useful for client/server interaction; one process provides service to a number of other processes. In this case, the mailbox is often referred to as a *port*. A **one-to-many** relationship allows for one sender and multiple receivers; it is useful for applications where a message or some information is to be broadcast to a set of processes. A **many-to-many** relationship allows multiple server processes to provide concurrent service to multiple clients.

The association of processes to mailboxes can be either static or dynamic. Ports are often statically associated with a particular process; that is, the port is created and permanently assigned to the process. Similarly, a one-to-one relationship is typically defined statically and permanently. When there are many senders, the association of a sender to a mailbox may occur dynamically. Primitives such as `connect` and `disconnect` may be used for this purpose.

A related issue has to do with the ownership of a mailbox. In the case of a port, it is typically owned and created by the receiving process. Thus, when the process is destroyed, the port is also destroyed. For the general mailbox case, the OS may offer a create mailbox service. Such mailboxes can be viewed either as being owned by the creating process, in which case they terminate with the process, or as being owned by the OS, in which case an explicit command will be required to destroy the mailbox.

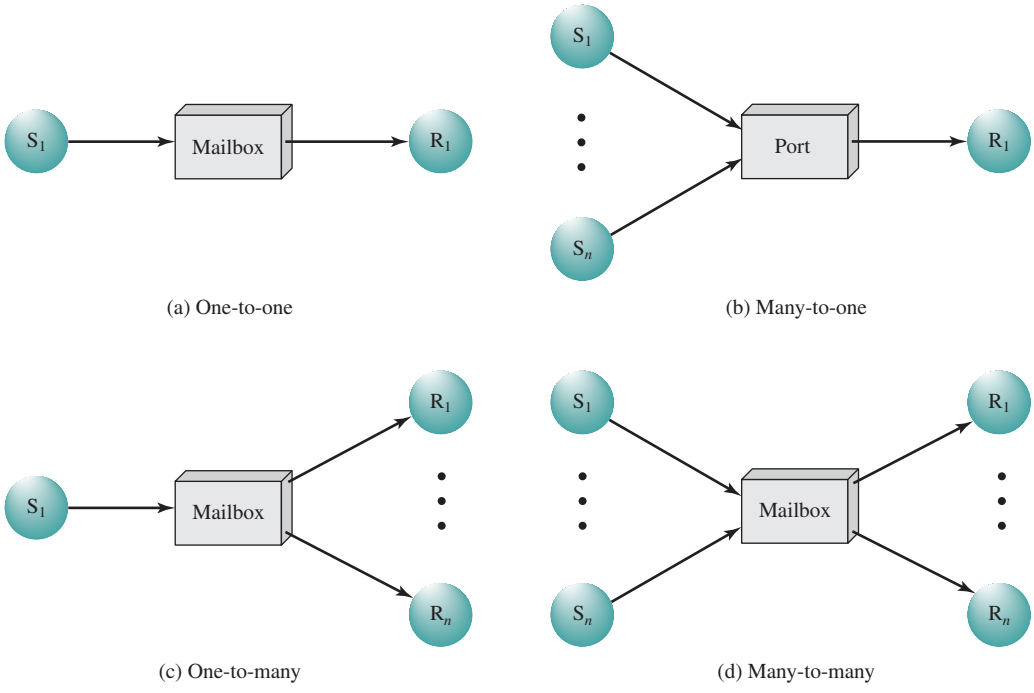


Figure 5.21 **Indirect Process Communication**

Message Format

The format of the message depends on the objectives of the messaging facility and whether the facility runs on a single computer or on a distributed system. For some operating systems, designers have preferred short, fixed-length messages to minimize processing and storage overhead. If a large amount of data is to be passed, the data can be placed in a file and the message then simply references that file. A more flexible approach is to allow variable-length messages.

Figure 5.22 shows a typical message format for operating systems that support variable-length messages. The message is divided into two parts: a header, which

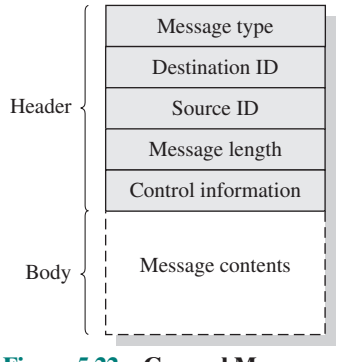


Figure 5.22 **General Message Format**

contains information about the message, and a body, which contains the actual contents of the message. The header may contain an identification of the source and intended destination of the message, a length field, and a type field to discriminate among various types of messages. There may also be additional control information, such as a pointer field so that a linked list of messages can be created; a sequence number, to keep track of the number and order of messages passed between source and destination; and a priority field.

Queueing Discipline

The simplest queueing discipline is first-in-first-out, but this may not be sufficient if some messages are more urgent than others. An alternative is to allow the specifying of message priority, on the basis of message type or by designation by the sender. Another alternative is to allow the receiver to inspect the message queue and select which message to receive next.

Mutual Exclusion

Figure 5.23 shows one way in which message passing can be used to enforce mutual exclusion (compare to Figures 5.4, 5.5, and 5.9). We assume the use of the blocking receive primitive and the nonblocking send primitive. A set of concurrent processes share a mailbox, box, which can be used by all processes to send and receive. The mailbox is initialized to contain a single message with null content. A process wishing to enter its critical section first attempts to receive a message. If the mailbox is empty, then the process is blocked. Once a process has acquired the message, it performs its critical section then places the message back into the mailbox. Thus, the message functions as a token that is passed from process to process.

```
/* program mutualexclusion */
const int n = /* number of process */
void P(int i)
{
      message msg;
      while (true) {
         receive (box, msg);
         /* critical section */;
         send (box, msg);
         /* remainder */;
      }
}
void main()
{
      create mailbox (box);
      send (box, null);
      parbegin (P(1), P(2), . . . , P(n));
}
```


VideoNote **Figure 5.23 Mutual Exclusion Using Messages**

The preceding solution assumes that if more than one process performs the receive operation concurrently, then:

- If there is a message, it is delivered to only one process and the others are blocked, or
- If the message queue is empty, all processes are blocked; when a message is available, only one blocked process is activated and given the message.

These assumptions are true of virtually all message-passing facilities.

As an example of the use of message passing, Figure 5.24 is a solution to the bounded-buffer producer/consumer problem. Using the basic mutual exclusion power of message passing, the problem could have been solved with an algorithmic structure similar to that of Figure 5.16. Instead, the program of Figure 5.24 takes advantage of the ability of message passing to be used to pass data in addition to signals. Two mailboxes are used. As the producer generates data, it is sent as messages to the mailbox `mayconsume`. As long as there is at least one message in that mailbox, the consumer can consume. Hence `mayconsume` serves as the buffer; the data in the buffer are organized as a queue of messages. The "size" of the buffer is determined by the global variable `capacity`. Initially, the mailbox `mayproduce`

```
const int
    capacity = /* buffering capacity */ ;
    null = /* empty message */ ;
int i;
void producer()
{   message pmsg;
    while (true) {
     receive (mayproduce,pmsg);
     pmsg = produce();
     send (mayconsume,pmsg);
    }
}
void consumer()
{   message cmsg;
    while (true) {
     receive (mayconsume,cmsg);
     consume (cmsg);
     send (mayproduce,null);
    }
}
void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1;i<= capacity;i++) send (mayproduce,null);
    parbegin (producer,consumer);
}
```

Figure 5.24 A Solution to the Bounded-Buffer Producer/Consumer Problem Using Messages

is filled with a number of null messages equal to the capacity of the buffer. The number of messages in `mayproduce` shrinks with each production and grows with each consumption.

This approach is quite flexible. There may be multiple producers and consumers, as long as all have access to both mailboxes. The system may even be distributed, with all producer processes and the `mayproduce` mailbox at one site and all the consumer processes and the `mayconsume` mailbox at another.

5.7 READERS/WRITERS PROBLEM

In dealing with the design of synchronization and concurrency mechanisms, it is useful to be able to relate the problem at hand to known problems, and to be able to test any solution in terms of its ability to solve these known problems. In the literature, several problems have assumed importance and appear frequently, both because they are examples of common design problems and because of their educational value. One such problem is the producer/consumer problem, which has already been explored. In this section, we will look at another classic problem: the readers/writers problem.

The readers/writers problem is defined as follows: There is a data area shared among a number of processes. The data area could be a file, a block of main memory, or even a bank of processor registers. There are a number of processes that only read the data area (readers) and a number that only write to the data area (writers). The conditions that must be satisfied are as follows:

1. Any number of readers may simultaneously read the file.
2. Only one writer at a time may write to the file.
3. If a writer is writing to the file, no reader may read it.

Thus, readers are processes that are not required to exclude one another, and writers are processes that are required to exclude all other processes, readers and writers alike.

Before proceeding, let us distinguish this problem from two others: the general mutual exclusion problem, and the producer/consumer problem. In the readers/writers problem, readers do not also write to the data area, nor do writers read the data area while writing. A more general case, which includes this case, is to allow any of the processes to read or write the data area. In that case, we can declare any portion of a process that accesses the data area to be a critical section and impose the general mutual exclusion solution. The reason for being concerned with the more restricted case is that more efficient solutions are possible for this case, and the less efficient solutions to the general problem are unacceptably slow. For example, suppose that the shared area is a library catalog. Ordinary users of the library read the catalog to locate a book. One or more librarians are able to update the catalog. In the general solution, every access to the catalog would be treated as a critical section, and users would be forced to read the catalog one at a time. This would clearly impose intolerable delays. At the same time,

it is important to prevent writers from interfering with each other, and it is also required to prevent reading while writing is in progress to prevent the access of inconsistent information.

Can the producer/consumer problem be considered simply a special case of the readers/writers problem with a single writer (the producer) and a single reader (the consumer)? The answer is no. The producer is not just a writer. It must read queue pointers to determine where to write the next item, and it must determine if the buffer is full. Similarly, the consumer is not just a reader, because it must adjust the queue pointers to show that it has removed a unit from the buffer.

We now examine two solutions to the problem.

Readers Have Priority

Figure 5.25 is a solution using semaphores, showing one instance each of a reader and a writer; the solution does not change for multiple readers and writers. The writer process is simple. The semaphore `wsem` is used to enforce mutual exclusion.

```
/* program readersandwriters */
int readcount;
semaphore x = 1,wsem = 1;
void reader()
{
    while (true){
     semWait (x);
     readcount++;
     if(readcount == 1)
         semWait (wsem);
     semSignal (x);
     READUNIT();
     semWait (x);
     readcount--;
     if(readcount == 0)
         semSignal (wsem);
     semSignal (x);
     }
}
void writer()
{
      while (true){
       semWait (wsem);
       WRITEUNIT();
       semSignal (wsem);
      }
}
void  main()
{
     readcount = 0;
     parbegin (reader,writer);
}
```

VideoNote **Figure 5.25 A Solution to the Readers/Writers Problem Using Semaphore: Readers Have Priority**

As long as one writer is accessing the shared data area, no other writers and no readers may access it. The reader process also makes use of `wsem` to enforce mutual exclusion. However, to allow multiple readers, we require that, when there are no readers reading, the first reader that attempts to read should wait on `wsem`. When there is already at least one reader reading, subsequent readers need not wait before entering. The global variable `readcount` is used to keep track of the number of readers, and the semaphore `x` is used to assure that `readcount` is updated properly.

Writers Have Priority

In the previous solution, readers have priority. Once a single reader has begun to access the data area, it is possible for readers to retain control of the data area as long as there is at least one reader in the act of reading. Therefore, writers are subject to starvation.

Figure 5.26 shows a solution that guarantees no new readers are allowed access to the data area once at least one writer has declared a desire to write. For writers, the following semaphores and variables are added to the ones already defined:

- A semaphore `rsem` that inhibits all readers while there is at least one writer desiring access to the data area
- A variable `writecount` that controls the setting of `rsem`
- A semaphore `y` that controls the updating of `writecount`

For readers, one additional semaphore is needed. A long queue must not be allowed to build up on `rsem`; otherwise writers will not be able to jump the queue. Therefore, only one reader is allowed to queue on `rsem`, with any additional readers queueing on semaphore `z`, immediately before waiting on `rsem`. Table 5.6 summarizes the possibilities.

Table 5.6 State of the Process Queues for Program of Figure 5.26

Readers only in the system	• *wsem* set • no queues
Writers only in the system	• *wsem* and *rsem* set • writers queue on *wsem*
Both readers and writers with read first	• *wsem* set by reader • *rsem* set by writer • all writers queue on *wsem* • one reader queues on *rsem* • other readers queue on *z*
Both readers and writers with write first	• *wsem* set by writer • *rsem* set by writer • writers queue on *wsem* • one reader queues on *rsem* • other readers queue on *z*

```
/* program readersandwriters */
int readcount,writecount; semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
   while (true){
     semWait (z);
          semWait (rsem);
               semWait (x);
                    readcount++;
                    if (readcount == 1)
                         semWait (wsem);
                    semSignal (x);
               semSignal (rsem);
          semSignal (z);
          READUNIT();
          semWait (x);
               readcount--;
               if (readcount == 0) semSignal (wsem);
          semSignal (x);
   }
}
void writer ()
{
   while (true){
     semWait (y);
          writecount++;
          if (writecount == 1)
               semWait (rsem);
     semSignal (y);
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
     semWait (y);
          writecount--;
          if (writecount == 0) semSignal (rsem);
     semSignal (y);
   }
}
void main()
{
   readcount = writecount = 0;
   parbegin (reader, writer);
}
```

VideoNote

Figure 5.26 A Solution to the Readers/Writers Problem Using Semaphore: Writers Have Priority

An alternative solution, which gives writers priority and which is implemented using message passing, is shown in Figure 5.27. In this case, there is a controller process that has access to the shared data area. Other processes wishing to access the data area send a request message to the controller, are granted access with an "OK" reply message, and indicate completion of access with a "finished" message. The controller is equipped with three mailboxes, one for each type of message that it may receive.

The controller process services write request messages before read request messages to give writers priority. In addition, mutual exclusion must be enforced. To do

```
void reader(int i)
{
    message rmsg;
        while (true) {
                rmsg = i;
            send (readrequest, rmsg);
            receive (mbox[i], rmsg);
            READUNIT ();
            rmsg = i;
            send (finished, rmsg);
        }
}
void writer(int j)
{
    message rmsg;
    while (true){
        rmsg = j;
        send (writerequest, rmsg);
        receive (mbox[j], rmsg);
        WRITEUNIT ();
        rmsg = j;
        send (finished, rmsg);
    }
}
```

```
void controller()
{
    while (true)
    {
        if (count > 0) {
            if (!empty (finished)) {
                receive (finished, msg);
                count++;
            }
            else if (!empty (writerequest)) {
                receive (writerequest, msg);
                writer_id = msg.id;
                count = count - 100;
            }
            else if (!empty (readrequest)) {
                receive (readrequest, msg);
                count--;
                send (msg.id, "OK");
            }
        }
        if (count == 0) {
            send (writer_id, "OK");
            receive (finished, msg);
            count = 100;
        }
        while (count < 0) {
            receive (finished, msg);
            count++;
        }
    }
}
```

VideoNote **Figure 5.27 A Solution to the Readers/Writers Problem Using Message Passing**

this the variable *count* is used, which is initialized to some number greater than the maximum possible number of readers. In this example, we use a value of 100. The action of the controller can be summarized as follows:

- If $count > 0$, then no writer is waiting and there may or may not be readers active. Service all "finished" messages first to clear active readers. Then service write requests, and then read requests.
- If $count = 0$, then the only request outstanding is a write request. Allow the writer to proceed and wait for a "finished" message.
- If $count < 0$, then a writer has made a request and is being made to wait to clear all active readers. Therefore, only "finished" messages should be serviced.

5.8 SUMMARY

The central themes of modern operating systems are multiprogramming, multiprocessing, and distributed processing. Fundamental to these themes, and fundamental to the technology of OS design, is concurrency. When multiple processes

are executing concurrently, either actually in the case of a multiprocessor system or virtually in the case of a single-processor multiprogramming system, issues of conflict resolution and cooperation arise.

Concurrent processes may interact in a number of ways. Processes that are unaware of each other may nevertheless compete for resources, such as processor time or access to I/O devices. Processes may be indirectly aware of one another because they share access to a common object, such as a block of main memory or a file. Finally, processes may be directly aware of each other and cooperate by the exchange of information. The key issues that arise in these interactions are mutual exclusion and deadlock.

Mutual exclusion is a condition in which there is a set of concurrent processes, only one of which is able to access a given resource or perform a given function at any time. Mutual exclusion techniques can be used to resolve conflicts, such as competition for resources, and to synchronize processes so they can cooperate. An example of the latter is the producer/consumer model, in which one process is putting data into a buffer, and one or more processes are extracting data from that buffer.

One approach to supporting mutual exclusion involves the use of special-purpose machine instructions. This approach reduces overhead, but is still inefficient because it uses busy waiting.

Another approach to supporting mutual exclusion is to provide features within the OS. Two of the most common techniques are semaphores and message facilities. Semaphores are used for signaling among processes and can be readily used to enforce a mutual exclusion discipline. Messages are useful for the enforcement of mutual exclusion and also provide an effective means of interprocess communication.

5.9 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

atomic	critical resource	mutual exclusion
binary semaphore	critical section	mutual exclusion lock (mutex)
blocking	deadlock	nonblocking
busy waiting	direct addressing	race condition
concurrency	general semaphore	semaphore
concurrent processes	indirect addressing	spin waiting
condition variable	livelock	starvation
coroutine	message passing	strong semaphore
counting semaphore	monitor	weak semaphore

Review Questions

5.1. List four design issues for which the concept of concurrency is relevant.
5.2. What are three contexts in which concurrency arises?
5.3. What is a race condition?
5.4. List three degrees of awareness between processes and briefly define each.