implement concurrency and none of them dominate the area (for example, see [Jamieson et al., 1987]). This chapter and Chapter 9 explore the intricacies of synchronization.

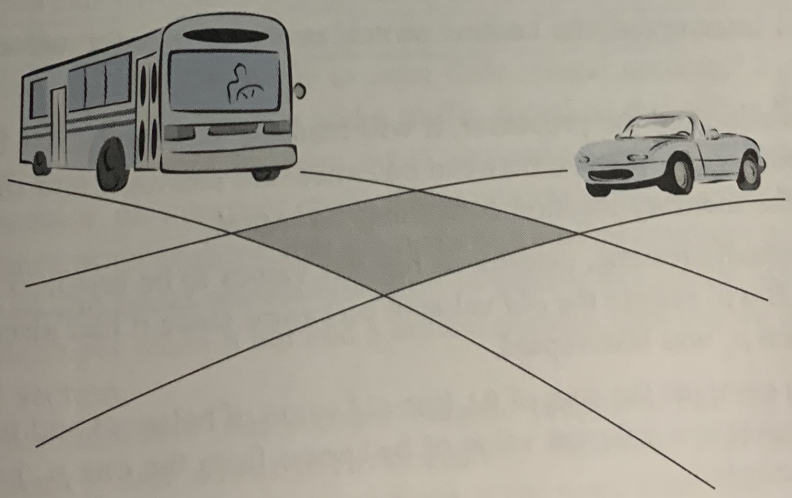
- Part of the difficulty with using synchronization mechanisms is in finding a good way for a high-level programming language to represent concurrency, then to fit the synchronization mechanism seamlessly into the concurrent programming language. Many widely accepted parallel programming languages do not address concurrency at all. The most modern languages such as Java and C# provide an extension for multiple active objects (a form of concurrency), but C and C++ do not support synchronization and concurrency at all.

CRITICAL SECTIONS

In automobile transportation, intersections are part of a street, but they are a unique part of the street because the intersection is a *shared* between two different streets (see Figure 8.3). In this cartoon, a bus proceeds along one street while a car moves along another. If the bus and the car get to the intersection at the same time, there will be a collision. We say that the intersection is a **critical section** of each street: It is perfectly acceptable for the bus or the car to use the intersection if the other is not currently using it. However, we can see that there will be a "transportation failure" if the bus and the car enter the intersection at the same time.

✦ FIGURE 8.3 Traffic Intersections

A traffic intersection is a critical section of these two streets, in the sense that only one vehicle can be in the intersection at a time. Either vehicle can use the critical section, provided that the other is not using it.



Critical sections occur in concurrent software whenever two processes/threads access a common shared variable. Like the bus and the car, there may be certain parts of the two processes that should not be executed concurrently. Such parts of the code are the software critical sections. For example, suppose that two processes, p_1 and p_2, execute concurrently to access a common integer variable, `balance`. For example, thread p_1 might handle credits to an account, while p_2 handles the debits. Both need access to the account `balance`

variable at different times (accessing `balance` is analogous to entering an intersection). This code will work exactly as expected most of the time: p_1 adds to the balance for a credit operation, and p_2 subtracts from the `balance` for a debit operation. However, disaster will strike if the two processes access the `balance` variable concurrently. The following code schema shows how the threads reference the shared `balance`:

```
shared double balance; /* shared variable */
```

Code schema for p_1

```
...
balance = balance+amount;
...
```

Code schema for p_2

```
...
balance = balance-amount;
...
```

These C language statements will be compiled into a few machine instructions, such as the following:

Code schema for p_1

```
load   R1, balance
load   R2, amount
add    R1, R2
store  R1, balance
```

Code schema for p_2

```
load   R1, balance
load   R2, amount
sub    R1, R2
store  R1, balance
```

Now suppose p_1 is executing the machine instruction

```
load   R2, amount
```

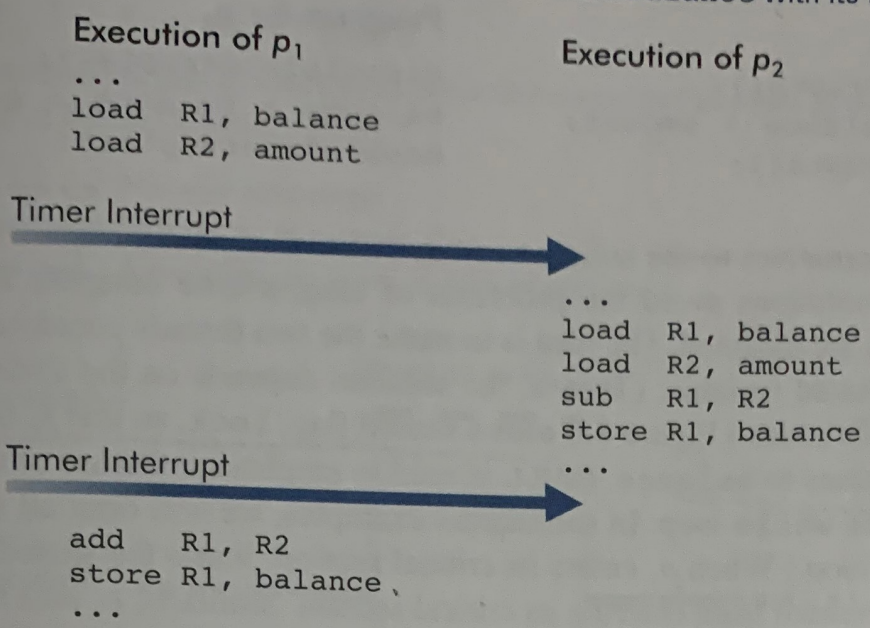
just as the interval timer expires. If the scheduler next selects thread p_2 to run and it executes its machine language code segment for the "`balance = balance-amount;`" instruction before p_1 regains control of the processor, then we will have the execution scenario shown in Figure 8.4. In this specific execution scenario (determined by *when* a timer interrupt occurred), the following sequence of actions take place:

- When p_1 is interrupted, the context switch saves its register values in its process descriptor.
- When p_2 is allocated the processor, it will read the same value of `balance` that p_1 read, compute the difference between `balance` and `amount`, and then store the difference at the memory location containing `balance`.
- p_1 will eventually resume, causing its register values to be restored from its process descriptor. It will restore the *old* value of `balance` since it had already been loaded into `R1` when p_1 was interrupted.
- p_1 will then compute the sum of `R1` (the old value of balance) and `R2` (the `amount`), and then generate a different value of `balance` from the one p_2 had written when p_1 was interrupted.
- The update of `balance` by p_2 will be lost!

The programs defining p_1 and p_2 each have a **critical section** with respect to their use of the shared variable `balance`. For p_1, the critical section is computing the sum of the `balance` and `amount`, but for p_2, the critical section is computing the difference of `balance` and `amount`. The concurrent execution of the two threads is not guaranteed to be **determinate**, *since not every execution of the two programs on the same data produces the same result.*

✦ FIGURE 8.4 A Critical Section

p_1 is interrupted as it finishes the load instruction. p_2 begins to execute, eventually entering the code block, where it subtracts the `amount` from the `balance`, storing the result back in memory. When p_1 resumes, it updates the old `balance` and overwrites `balance` with its result.



We say that there is a **race condition** between p_1 and p_2, because the outcome of the computation depends on the relative times that the two processes execute their respective critical sections. If this race results in a "close finish," with the two processes executing their respective critical sections at the same time, the computation may be faulty.

It is not possible to detect a critical section problem (or race condition) by considering only p_1's program or only p_2's program. The problem occurs because of sharing, not because of any error in the sequential code. The critical section problem can be avoided by having either thread enter its corresponding critical section any time it needs to do so *except when the other thread is currently in its critical section.*

How can the two threads cooperate to enter their critical sections? In the traffic intersection case (see Figure 8.3), we can add a traffic signal so that either the bus or the car can proceed (but not both at the same time), depending on the signal. In a multiprogrammed uniprocessor, an interrupt enabled one process to be stopped and another to be started. If the programmer realized that the occurrence of an interrupt could lead to erroneous results, he or she could control interrupts to behave like traffic lights: The program would *disable interrupts* when it entered a critical section, then enable them when it finished the critical section.

Figure 8.5 illustrates how the account balance programs can be coded using the `enableInterrupt()` and `disableInterrupt()` function: This solution does not allow both threads to be in their critical sections at the same time. Interrupts are disabled on entry to the critical section and then enabled upon exit. Unfortunately, this technique may affect the behavior of the I/O system because interrupts are disabled for an arbitrarily long time (as determined by an application program). In particular, suppose a program contained an infinite loop inside its critical section. The interrupts would be permanently disabled. For this reason, user mode programs cannot invoke `enableInterrupt()` and `disableInterrupt()`.

✦ FIGURE 8.5 Disabling Interrupts to Implement the Critical Section

Interrupts are disabled while a process enters its critical section, then enabled when it leaves.

```
shared double amount, balance;    /* Shared variables */
```

Program for p_1

```
disableInterrupts();
balance = balance + amount;
enableInterrupts();
```

Program for p_2

```
disableInterrupts();
balance = balance - amount;
enableInterrupts();
```

There are alternatives to the solution in Figure 8.5 that do not require interrupts to be disabled. Such solutions avoid the problems of long/infinite compute intervals during which interrupts are disabled. The idea is to make the two threads coordinate their actions using another shared variable. (That is, the solution depends on the ability of the OS to provide shared variables.) Figure 8.6 uses a shared flag, `lock`, so that p_1 and p_2 can coordinate their accesses to `balance`. (`NULL` is used to emphasize the use of a null statement in the body of the `while` loop. In subsequent examples, we will omit all statements from the body of the loop.) When p_1 enters its critical section, it sets the shared `lock` variable, so p_2 will be prevented from entering its critical section. Similarly, p_2 uses the `lock` to prevent p_1 from entering its critical section at the wrong time.

✦ FIGURE 8.6 Critical Sections Using a Lock

In this solution, the `lock` variable coordinates the way the two processes enter their critical sections. On critical section entry, a process waits while `lock` is `TRUE`.

```
shared boolean lock = FALSE;      /* Shared variables */
shared double amount, balance;    /* Shared variables */
```

Program for p_1

```
...
/* Acquire lock */
 while(lock) {NULL;};
 lock = TRUE;
/* Execute crit section */
 balance = balance + amount;
/* Release lock */
 lock = FALSE;
 ...
```

Program for p_2

```
...
/* Acquire lock */
while(lock) {NULL;};
lock = TRUE;
/* Execute crit section */
 balance = balance - amount;
/* Release lock */
lock = FALSE;
...
```

Figure 8.7 illustrates an execution pattern in which the two threads compete for the critical section: Suppose p_1 is interrupted during the execution of the statement

```
balance = balance + amount;
```

after having set lock to `TRUE`. Suppose p_2 then begins to execute. p_2 will wait to obtain the `lock` (and to enter its critical section) at its `while` statement. Eventually, the clock interrupt will interrupt p_2 and resume p_1, which can complete its critical section. Potentially p_2's entire timeslice is spent executing the `while` statement. When p_1 is running on the processor and executes

```
lock = FALSE;
```

problems caused by disabling the interrupts for an extended period of time since an interrupt will never be delayed for more than the time taken to execute the `while` statement.

✦ FIGURE 8.8 Lock Manipulation as a Critical Section

The `enter()` system call uses the `while` statement to wait for a critical section to become available, but it will only disable interrupts for a few machine instructions before enabling them again. This will prevent interrupts from being delayed for more than a few instruction executions.

```
enter(lock) {
  disableInterrupts();
/* Wait for lock */
  while(lock) {
    /* Let interrupt occur */
    enableInterrupts();
    disableInterrupts();
  }
  lock = TRUE;
  enableInterrupts();
}
```

```
exit(lock) {
  disableInterrupts();
  lock = FALSE;
  enableInterrupts();
```

The `enter()` and `exit()` system calls can be used to solve the general critical section problem (although we will study a more general mechanism in Section 8.3). Here is how they can be used to solve the `balance` manipulation problem:

```
shared double amount, balance;    /* Shared variables */
shared int lock = FALSE;          /* Synchronization variable */
```

Program for p_1

```
enter(lock);
balance = balance + amount;
exit(lock);
```

Program for p_2

```
enter(lock);
balance = balance - amount;
exit(lock);
```

DEADLOCK

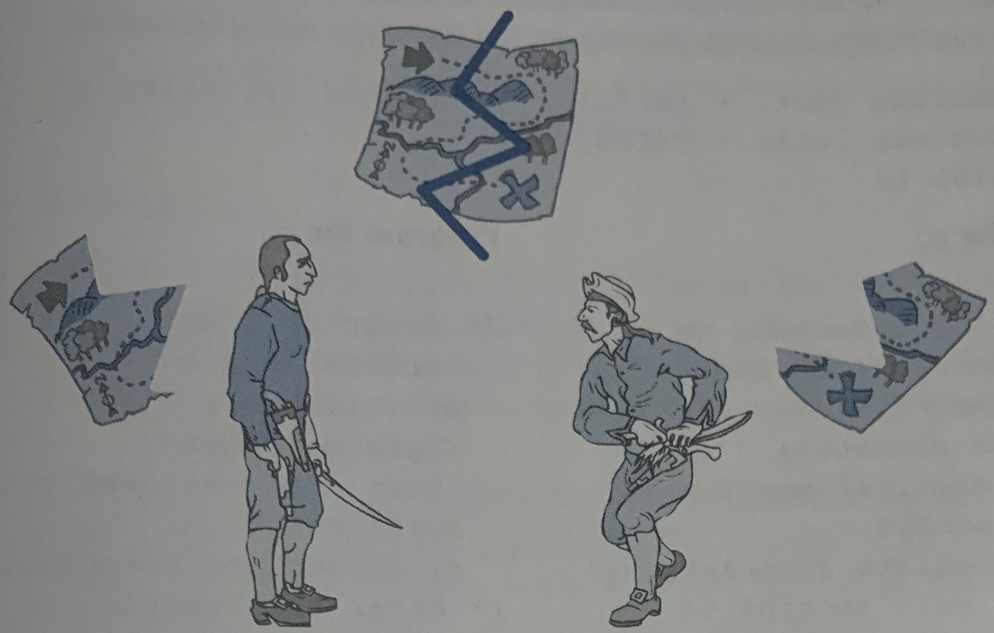
Critical sections are fundamental to concurrent programming since they are the parts of the computation during which individual processes manipulate shared resources (such as a variable). The existence of critical sections creates an environment in which a new, subtle problem can occur: **deadlock**. In a deadlock situation, two or more processes/threads get into a state whereby each is controlling a resource that the other needs. For example, suppose two pirates have each obtained half of a treasure map (see Figure 8.9). Each pirate needs the other half of the map to obtain the treasure, but neither will give up his half of the map. This is a deadlock.

In software, deadlocks occur because one process holds a resource (such as file A) while requesting another (such as file B). At the same time, another process holds the second resource (file B) while requesting the first one (file A). Since a request operation blocks the calling process until that resource is allocated, neither process will ever have all its desired resources allocated to it and both will remain in this **deadlock state** forever.

Here is another concrete example: Suppose there are two threads, p_1 and p_2, manipulating a common list. Each task is able to add or delete an entry, requiring that the list

FIGURE 8.9 Deadlocked Pirates

Each pirate holds half of a treasure map, but needs to acquire the other half to find the treasure.



length in the list header be updated. That is, when a delete operation occurs, the length must be decremented, and when an entry is added to the list, the length must be incremented. To ensure consistency between the list and its header, we could first try the approach shown in Figure 8.10 (although this will turn out to be an incorrect solution).

If threads p_1 and p_2 are executed concurrently:

- A clock interrupt may occur after p_1 deletes an element but before it updates the length in the list descriptor.
- If p_2 adds an element to the list and updates the length before p_1 resumes, then the contents of the list and its length in the descriptor will not be consistent with one another.

In this example, a process *should update both the list and the descriptor or neither*. So we try a different solution in order to place modifications to the list and the descriptor within a more complex critical section scheme, as shown in Figure 8.11.

- When p_1 enters its critical section to manipulate the list, it sets `lock1`.
- Thus p_2 will be prevented from entering its critical section to manipulate the list when it tests `lock1`.
- The same also holds for p_2 when it enters its list manipulation critical section and for both p_1 and p_2 to update the length.
- Suppose p_1 is interrupted during the `<intermediate computation>` (after having set `lock1` to `TRUE`) and p_2 begins to execute.
- p_2 will set `lock2` and then wait for `lock1` at the while statement.
- Eventually, the clock interrupt will resume p_1, and p_1 can then complete the `<intermediate computation>` and block at its `while` statement test on `lock2` (prior to updating the descriptor).

✦ FIGURE 8.13 Example Concurrent Processes

These code fragments represent the activity of two different concurrent processes. `proc_A` writes a shared variable, *x*, and reads a shared variable, *y*. `proc_B` reads *x* and writes *y*.

```
shared double x, y; /* Shared variables */
proc_A() {
  while(TRUE) {
    <compute A1>;
    write(x); /* Produce x */
    <compute A2>;
    read(y); /* Consume y */
  }
}
```

```
proc_B() {
  while(TRUE) {
    read(x); /* Consume x */
    <compute B1>;
    write(y); /* Produce y */
    <compute B2>;
  }
}
```

8.3 • SEMAPHORES: THE BASIS OF MODERN SOLUTIONS

Busy traffic intersections address the critical section problem by adding a semaphore—a traffic light—to coordinate use of the shared intersection. In software, a **semaphore** is an OS abstract data type that performs operations similar to the traffic light. The semaphore will allow one process (such as the car) to control the shared resource while the other process (such as the bus) waits for the resource to be released. Before discussing the principles of operation of semaphores, let's consider the assumptions under which semaphores operate.

An acceptable solution to the critical section problem has these requirements:

- Only one process at a time should be allowed to be executing in its critical section (**mutual exclusion**).
- If a critical section is free and a set of processes all want to enter the critical section, then the decision about which process should be chosen to enter the critical section should be made by the collection of processes instead of by an external agent (such as an arbiter or scheduler).
- If a process attempts to enter its critical section and the critical section becomes available, then the waiting process cannot be blocked from entering the critical section for an indefinite period of time.
- Once a process attempts to enter its critical section, then it cannot be forced to wait for more than a bounded number of other processes to enter the critical section before it is allowed to do so.

For purposes of discussion, this section highlights other important aspects of the problem by considering the two process skeletons shown in Figure 8.14. In this and subsequent figures, the statement

```
fork(proc, N, arg1, arg2, ..., argN)
```

means that a single-threaded process is created and begins executing `proc()` in its own address space using the *N* arguments provided. `<shared global declarations>` are

intended to be the shared variables accessible in the address space of all the processes. (The order in which the procedure and shared global variables is declared is unspecified. Here the variables are declared after the procedures, but we often declare them before the procedures.)

✦ FIGURE 8.14 Cooperating Processes

This is the format for describing multiple processes or multiple threads in a process. In this example, two processes are created, one executing `proc_0()`, and the other executing `proc_1()`.

```
proc_0() {
  while(TRUE) {
    <compute section>;
    <critical section>;
  }
}
<shared global declarations>;
<initial processing>;
fork(proc_0, 0);
fork(proc_1, 0);
```

```
proc_1() {
  while(TRUE) {
    <compute section>;
    <critical section>;
  }
}
```

The following assumptions are made about the execution of the software schema in the figure:

- Writing and reading a memory cell common to the two processes/threads is an indivisible operation. Any attempt by the two processes to execute simultaneous memory read or write operations will result in some unknown serial ordering of the two operations, but the two operations will not happen at the same time.
- The threads are not assumed to have any priority, where one or the other would take precedence in the case of simultaneous attempts to enter a critical section.
- The relative speeds of the threads are unknown, so one cannot rely on speed differentials (or equivalence) in arriving at a solution.
- As indicated in Figure 8.14, the individual process executions are assumed to be sequential and cyclic.

PRINCIPLES OF OPERATION

Edsger Dijkstra was well-known as the inventor of the semaphore. It was the first software-oriented primitive to accomplish process synchronization [Dijkstra, 1968]. Over 35 years ago, Dijkstra's work on semaphores established the foundation of modern techniques for accomplishing synchronization. It is still a viable approach to managing communities of cooperating processes. Dijkstra's classic paper accomplished many things:

- Introducing the idea of *cooperating sequential processes*
- Illustrating the difficulty in accomplishing synchronization using only conventional (at that time) machine instructions
- Postulating the primitives

- Proving that they worked
- Providing a number of examples (many of which are used in examples and exercises in this book).

At the time of Dijkstra's work, classic (single-threaded) processes were used to represent computation; threads were not invented for another twenty years. Dijkstra semaphores are described in terms of classic processes, although they apply to threads as well. In Dijkstra's original paper, the P operation was an abbreviation for the Dutch word *proberen*, meaning "to test," and the V operation was an abbreviation for *verhogen*, meaning "to increment."

A semaphore, *s*, is a nonnegative integer variable changed or tested only by one of two indivisible access routines:

```
V(s): [s = s + 1]
P(s): [while(s == 0) {wait}; s = s - 1]
```

The square braces surrounding the statements in the access routines indicate that the operations are **indivisible**, or **atomic**, operations. That is, all statements between the "[" and "]" are executed as if they were a single machine instruction. More precisely, the process executing the V routine cannot be interrupted until it has completed the routine. The P operation is more complex. If *s* is greater than 0, it is tested and decremented as an indivisible operation. However, if *s* is equal to 0, the process executing the P operation *can* be interrupted when it executes the `wait` command in the range of the `while` loop. The indivisible operation applies only to the test and control flow after the test, not to the time the process waits due to *s*'s being equal to zero.

The P operation is intended to indivisibly test an integer variable and to block the calling process if the variable is not positive. The V operation indivisibly signals a blocked process to allow it to resume operation. As our first semaphore example, let's reconsider the account balance code from Section 8.1 (see Figure 8.15). The initial value of the semaphore named `mutex` is 1 ("mutex" is a classic name from Dijkstra's original paper meaning "mutual exclusion"). When a process gets ready to enter its critical section, it applies the P operation to `mutex`. The first process to invoke P on `mutex` passes and the second blocks. When the first invokes the V operation on `mutex`, it continues to execute, thus enabling the second to proceed when it gets control of the CPU.

Next, we'll consider a series of examples that use semaphores to solve the critical section problem and to synchronize the operation of two processes or threads. We start with some simple examples using **binary semaphores**, meaning that the value of the semaphore takes on only the values 0 and 1. Usually, but not always (see the example), the semaphore is initialized to have the value 1.

✦ FIGURE 8.15 Semaphores on the Shared Balance Problem

The semaphore solution initializes the semaphore, `mutex`, to be one. Each process invokes the P operation to enter the critical section, and the V operation when it leaves the critical section.

```
proc_0() {
  ...
/* Enter critical section */
  P(mutex);
    balance = balance + amount;
/* Exit critical section */
  V(mutex);
  ...
}
semaphore mutex = 1;
fork(proc_0, 0);
fork(proc_1, 0);
```

```
proc_1() {
  ...
/* Enter critical section */
  P(mutex);
    balance = balance - amount;
  /* Exit critical section */
  V(mutex);
  ...
  }
```

Using Semaphores

Various classic synchronization problems were introduced in Dijkstra's original paper, and many appeared in later papers and textbooks. This set of examples reviews several of those that illustrate recurring synchronization scenarios.

The Basic Synchronizing Problem

You have seen how semaphores are used to solve the critical section problem in the account balance example. In Figure 8.13 you saw that there is another kind of synchronization problem where one process needs to coordinate with another by sending it a signal. The solution shown in Figure 8.16 illustrates how semaphores are used for this type of synchronization. We cannot simply substitute enable/disable calls for P and V in this example in order to return to the original solution (as we could for the account balance example) because more than one semaphore is used for the synchronization. In this case, the semaphore is used to exchange synchronization signals among processes, as opposed to solving the critical section problem.

Software/Hardware Device Interaction

In Chapter 4, you learned about the software/hardware interface between a device driver and controller. The `busy` and `done` flags in the status register can be viewed as hardware implementations of semaphores, since they are used to synchronize the operation of the software driver and the hardware controller. Figure 8.17 is a code skeleton representing the interaction.

✦ FIGURE 8.16 Using Semaphores to Synchronize Two Processes

The proc_A and proc_B processes need to coordinate their activity so that proc_B does not attempt to read x until after proc_B has written it. Conversely, proc_A should not attempt to read y until after proc_B has written a new value to y. proc_A uses semaphore s1 to signal proc_B, and proc_B uses s2 to signal proc_A.

```
proc_A() {
  while(TRUE) {
    <compute A1>;
    write(x); /* Produce x */
    V(s1); /* Signal proc_B */
    <compute A2>;
 /* Wait for proc_B signal */
    P(s2);
    read(y); /* Consume y */
  }
}
semaphore s1 = 0;
semaphore s2 = 0;
fork(proc_0, 0);
fork(proc_1, 0);
```

```
proc_B() {
  while(TRUE) {
/* Wait for proc_A signal */
    P(s1);
    read(x); /* Consume x */
    <compute B1>;
    write(y); /* Produce y */
    V(s2); /* Signal proc_A */
    <compute B2>;
  }
}
```

✦ FIGURE 8.17 The Driver-Controller Interface Behavior

The busy and done hardware flags are used like signaling semaphores. The driver coordinates with the controller by setting busy, and the controller coordinates its state with the driver using the done flag.

```
/* Map the hardware flags to shared semaphores */
semaphore busy = 0, done = 0;
driver() { /* Synchronization behavior of the driver */
  <preparation for device operation>;
  V(busy); /* Start the device */
  P(done); /* Wait for the device to complete */
  <complete the operation>;
}
controller() { /* Controller's hardware loop */
  while(TRUE) {
    P(busy); /* Wait for a start signal */
    <perform the operation>;
    V(done); /* Tell driver that hardware has completed */
  }
}
```

The code fragment in this figure only models the synchronization behavior of the device driver software and the hardware controller, not all the functional behavior of the device driver and controller. (For example, if this solution were the basis of an implementation, it would cause the calling process to block—which an implementation would not

do.) In the model, `busy` and `done` are initialized to 0, so when the device controller is started, it enters an endless loop in which it synchronizes its operation with the software process by testing the `busy` flag. If `busy` is 0—the initial condition—the controller blocks waiting for a signal from the driver. An application program calls the driver whenever it wants to perform an I/O operation. After preparing for the operation (for example, by setting controller registers, device status table entries, and so on), it signals the hardware process by a V operation on the `busy` semaphore. The V operation unblocks the controller and then blocks the driver on the `done` semaphore. When the device has completed the operation, it signals the driver by a V operation on the `done` flag.

THE BOUNDED BUFFER (PRODUCER–CONSUMER) PROBLEM

The bounded buffer problem occurs regularly in concurrent software. In Figure 5.11 we saw how "The Pure Cycle Water Company" illustrates the way buffers are used. Dijkstra used this problem to demonstrate different uses of semaphores in the same problem [Dijkstra, 1968]. Suppose a system incorporates two single-threaded (classic) processes, one of which produces information (the *producer* process) and another that uses the information (the *consumer* process). The two processes communicate by having the producer obtain an empty buffer from an empty buffer pool, fill it with information, and place it in a pool of full buffers. The consumer obtains information by picking up a buffer from the full buffer pool, copying the information from the buffer, and placing the buffer into the empty buffer pool for recycling. The producer and consumer use a fixed, finite number, N, of buffers to pass an arbitrary amount of information between them. This solution uses the buffers to keep the producer and consumer roughly synchronized.

Figure 8.18 is a program schema for the producer and consumer processes. The `empty` and `full` semaphores illustrate a new type of semaphore, called a **general** semaphore (it is also often called a **counting** semaphore). Whereas a binary semaphore takes on only the values 0 and 1, the counting semaphore takes on values from 0 to N for the N-buffer problem. In the solution, the counting semaphores serve a dual purpose. They keep a count of the number of empty and full buffers. They also are used to synchronize the process operation by blocking the producer when there are no empty buffers and blocking the consumer when there are no full buffers.

The buffers are a block of memory logically split into N parts. Each buffer must contain space for links to associate the buffer with other empty or full buffers and space for the data itself. Since the producer and consumer each manipulate these links, the code for buffer pool manipulation must be treated as a critical section. The `mutex` semaphore protects access to the two buffer pools so that only one process takes or puts a buffer at a time. The `V()` operations signal the release of a buffer to the empty or full pool of buffers.

The `mutex` semaphore is used to protect the critical section relating to manipulating buffers (such as list insert/delete operations). The `P(empty)` operation blocks the producer if there are no empty buffers. Similarly, the `P(full)` operation blocks the consumer if there are no full buffers.

✦ FIGURE 8.18 The Bounded Buffer Problem

This solution uses three semaphores: `mutex` is a binary semaphore, while `full` and `empty` are general semaphores (taking on values from 0 to *N*. The `mutex` semaphore protects the critical section related to adding/deleting buffers to/from a pool. The two general semaphores are used by the producer/consumer process to tell the other process that there are full/empty buffers available.

```
producer() {
  bufType *next, *here;
  while(TRUE) {
    produceItem(next);
  /*Claim an empty buffer */
    P(empty);
    /* Manipulate the pool */
    P(mutex);
    here = obtain(empty);
    V(mutex);
    copyBuffer(next, here);
    /* Manipulate the pool */
      P(mutex);
      release(here, fullPool);
      V(mutex);
    /* Signal a full buffer */
      V(full);
  }
}
```

```
consumer() {
  bufType *next, *here;
  while(TRUE) {
  /* Claim a full buffer */
    P(full);
  /* Manipulate the pool */
    P(mutex);
      here = obtain(full);
    V(mutex);
    copyBuffer(here, next);
  /* Manipulate the pool */
    P(mutex);
      release(here, emptyPool);
    V(mutex);
  /* Signal an empty buffer */
    V(empty);
    consumeItem(next);
  }
}
```

```
semaphore mutex = 1;
semaphore full = 0;
semaphore empty = N;
bufType buffer[N];
fork(producer, 0);
fork(consumer, 0);
```

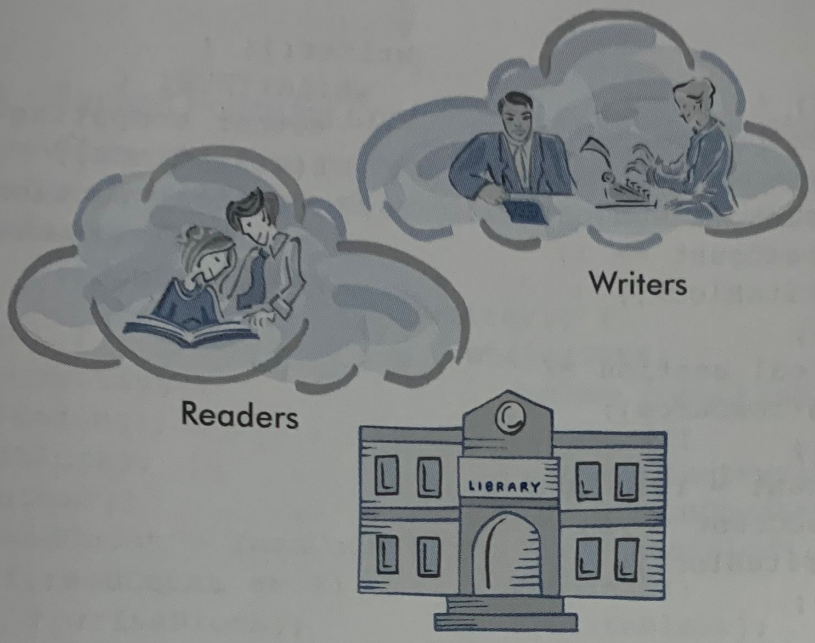
THE READERS–WRITERS PROBLEM

Courtois, Heymans, and Parnas [1971] posed an interesting, recurring synchronization problem called the readers-writers problem. Again, since this problem was posed using classic, single-threaded processes, the example also uses them; the solution applies to multithreaded computations as well. Suppose a resource is to be shared among a community of processes of two distinct types: readers and writers. A **reader** process can share the resource with any other reader process, but not with any writer process. A **writer** process requires exclusive access to the resource whenever it acquires any access to the resource.

This scenario is similar to one in which a file is to be shared among a set of processes (see Figure 8.19). If a process wants only to read the file, then it may share the file with any other process that also only wants to read the file. If a writer wants to modify the file, then no other process should have access to the file while the writer has access to it.

✦ **FIGURE 8.19** The Readers–Writers Problem

Readers and writers compete for the shared resource (access to a book in the cartoon). Readers can share the resource, but each writer must have exclusive control of the resource.



Several different policies could be implemented for managing the shared resource. For example, as long as a reader holds the resource and there are new readers arriving, any writer must wait for the resource to become available. The algorithm shown in Figure 8.20 illustrates how this policy is implemented. In this policy, the first reader accessing the shared resource must compete with any writers, but once a reader succeeds, other readers can pass directly into the critical section, provided that at least one reader is still in the critical section. The readers keep a count of the number of readers in the critical section, with the `readCount` variable, which is updated and tested inside its own critical section. Only the first reader executes the `P(writeBlock)` operation, while every writer does so, since every writer must compete with the first reader. Similarly, the last reader to yield the critical section must perform the V operation on behalf of all readers that accessed the shared resource.

While this solution implements the first policy, it is easy to see that the policy may not produce the desired result. Readers can dominate the resource so that no writer ever gets a chance to access it. This situation is analogous to the case in which a pending update of a file must wait until all reads have completed.

In most cases, you would like the updates to take place as soon as possible. This preference leads to an alternative policy that favors writers. That is, when a writer process requests access to the shared resource, any subsequent reader process must wait for the writer to gain access to the shared resource and then release it.

✦ **FIGURE 8.20** First Policy for Coordinating Readers and Writers

In this policy, readers have priority over writers. This is true since once a reader gets control of the shared resource, a stream of subsequent readers can block any writer for an indefinite period of time.

```
reader() {
  while(TRUE) {
    <other computing>;
    P(mutex);
      readCount = readCount+1;
      if (readCount == 1)
        P(writeBlock);
    V(mutex);
    /* Critical section */
      access(resource);
    P(mutex);
      readCount = readCount-1;
      if(readCount == 0)
        V(writeBlock);
    V(mutex);
  }
}
resourceType *resource;
int readCount = 0;
semaphore mutex = 1;
semaphore writeBlock = 1;
/* Start the readers and writers */
fork(reader, 0); /* Could be many */
fork(writer, 0); /* Could be many */
```

```
writer(){ {
  while(TRUE) {
    <other computing>;
    P(writeBlock);
  /* Critical section */
      access(resource);
    V(writeBlock);
  }
}
```

An algorithm to implement the second policy is shown in Figure 8.21. This policy still allows a stream of readers to enter the critical section until a writer arrives. Once a writer arrives, it takes priority over all subsequent readers, except those already accessing the shared resource. When the first writer arrives, it will obtain the `readBlock` semaphore. Then it blocks on the `writeBlock` semaphore, waiting for all readers to clear the critical section. The next reader to arrive will obtain the `writePending` semaphore and then block on the `readBlock` semaphore. Suppose another writer arrives at this time. It will block on the `writeBlock` semaphore, assuming the first writer has progressed to the critical section. If a second reader arrives, it will block at the `writePending` semaphore. When the first writer leaves the critical section, any subsequent writer is required to have priority over all readers. The second and subsequent writers are blocked at `writeBlock`, and no reader is blocked on the semaphore, so the writers will dominate the resource. When all writers have completed, the readers are then allowed to use the resource.

This example highlights a new problem. Semaphores provide an abstraction of hardware-level synchronization into a software mechanism used to solve simple problems, but complex problems like the readers-writers problem are more difficult. How do we know a solution such as the second readers-writers solution is correct? We are left with two choices:

- Create a higher-level abstraction (you will learn more about this in Chapter 9).
- Prove that our synchronized program is correct. That is, semaphores have not eliminated the need for proofs, but they enable us to write more complex scenarios than we could without them.

✦ FIGURE 8.21 Second Policy for Coordinating Readers and Writers

The second readers-writers policy gives priority to writers. Even if there are readers using the shared resource, when a writer arrives, it will obtain access to the resource before any other readers are allowed access.

```
reader() {
  while (TRUE) {
    <other computing>;
    P(writePending);
      P(readBlock);
        P(mutex1);
          readCount = readCount+1;
          if(readCount == 1)
            P(writeBlock);
        V(mutex1);
      V(readBlock);
    V(writePending);
      access(resource);
    P(mutex1);
      readCount = readCount-1;
      if(readCount == 0)
        V(writeBlock);
    V(mutex1);
  }
}
```

```
writer() {
  while(TRUE) {
    <other computing>;
    P(mutex2);
      writeCount=writeCount+1;
      if(writeCount==1)
        P(readBlock);
    V(mutex2);
    P(writeBlock);
      access(resource);
    V(writeBlock);
    P(mutex2);
      writeCount=writeCount-1;
      if(writeCount==0)
        V(readBlock);
    V(mutex2);
  }
}
```

```
resourceType *resource;
int readCount = 0, writeCount = 0;
semaphore mutex1 = 1, mutex2 = 1;
semaphore readBlock = 1;
semaphore writePending = 1;
semaphore writeBlock = 1
/* Start the readers and writers */
fork(reader, 0); /* Could be many */
fork(writer, 0); /* Could be many */
```

PRACTICAL CONSIDERATIONS

There are important practical considerations related to implementing semaphores. The rest of this section deals with how to implement semaphores, avoid busy-waiting on semaphores, and view semaphores as resources. We also consider an important detail related to the implementation of the V operation: active versus passive behavior.