Multiprogramming created an environment for concurrent classic processes. It also made it possible for a programmer to create a group of cooperating processes to work concurrently on a single problem. The resulting computation could even exhibit true parallelism if some of the processes were performing I/O operations while another one of the processes was using the processor. If the cooperating processes were executing on a multiprocessor, then multiple processes in the group could be executing in parallel. The threads refinement made it possible for a single process to take advantage of parallelism between the processor and the I/O and across CPUs on a multiprocessor. However, multiple cooperating threads introduce the potential for new synchronization problems in software implementations: deadlock, critical sections, and nondeterminacy. These synchronization problems can occur whenever two or more concurrent processes/threads use any shared resource. In this chapter, we will first see how synchronization problems arise in concurrent applications. Then, we will look at an abstract mechanism that can be used to solve synchronization problems. Finally, we will discuss ways that an OS can implement the abstract mechanisms.

8.1 • Cooperating Processes

As you have learned in your early programming courses, a program is a realization of a serial algorithm—it is a step-by-step procedure for accomplishing some information processing task. The science (and art) of designing serial algorithms to specify computations has dominated programming for over half a century. Consequently, computing environments have focused on supporting serial computation. A classic process and a modern...
thread are both sequential abstractions to execute a serial algorithm. While popular programming languages such as C and C++ have generally neglected to address concurrency, the underlying hardware technology has systematically moved toward it, using distributed and parallel computing machines. Economic pressures have driven application requirements toward the use of parallel and distributed hardware. This trend is apparent in contemporary management information systems, office computing environments, and numerical applications.

We often rely on synchronization when we meet with other people. Suppose Betty, John, and Pat decide to have a meeting in Betty’s office. They must next agree on a time for the meeting. They each consult their calendars to decide when all of them wish to meet. By reaching an agreement about the time of the meeting, they are explicitly committing to synchronize their independent schedules so that they will all be in the same place at the same time. In an espionage movie, synchronization can be finer grained: Suppose that a team has a plan for attacking a fort in which each member of the team must be prepared to perform a specific task at exactly the same time, say 5:00, as all the others perform their own specific tasks. In this case, it is important that the team members all perform their actions at almost exactly the same time. Just before the team members start their mission, say at 1:00, they synchronize their watches by setting them to 1:00 at exactly the same moment. This provides the best assurance that 4 hours later, they will all perform their specific tasks at the same instant.

In the software context, synchronization refers to the act of ensuring that independent processes/threads begin to execute a designated block of code at the same logical time. Now let’s see how synchronization manifests itself in concurrent software. You first saw concurrent software in the UNIX and Windows examples in Chapter 2. In the UNIX example, you saw how to use a parent process to implement a shell command line interpreter, and a child to execute each command. Here is the main loop of that code:

```c
while(TRUE) {
  ...
  // Create a process to execute the command
  if((chPID = fork()) == 0) {
    // This is the child
    execv(command.name, command.argv);
  }
  // Wait for the child to terminate
  thisChPID = wait(&stat);
}
```

In the Windows example, the parent process launches the child process, but it does not wait for the child to finish before launching the next child. Here is the main loop of that code:

```c
while (fgets(cmdLine, MAX_LINE_LEN, fid) != NULL) {
  // b. Create a new process to execute the command
  if (!CreateProcess(NULL, cmdLine, ...) {
    /* error handling code ... */
  }
}
Figure 8.1 is a graph representing the code skeletons for both Laboratory Examples: Each large circle represents a block of code (such as “Execute Command”). The small circles represent places where a thread of execution can branch one way or the other (such as “Another Command?”). The arrows show the control flow among the blocks of code in the circles. Concurrency is represented by a case where multiple arrows leave a large circle (such as for the “fork() code” circle). The “Wait for Child to Terminate” large circle represents a case where concurrent control flow merges back into serial execution (two arrows coming into the large circle, but just one leaving it).

**FIGURE 8.1** Command Execution

Part (a) represents the control flow of the UNIX code segment. The parent process synchronizes with the child process activity by waiting for each child to terminate before starting the next one. Part (b) is the control flow pattern for the Windows code segment. In this case, the parent creates a child, but does not synchronize with that child again. Instead, the parent creates the next child to execute the next command.

In Figure 8.1(a) we see that the parent program (the shaded blue circles) creates a child process to execute a command, then waits for the child to terminate before reading the next command. When this program processes 5 commands, the 5 processes that execute the commands will be created sequentially. There is concurrency between the parent and at most one child at a time.

In the Windows program shown in Figure 8.1(b), the parent process creates a process to execute a command, then immediately goes back to the top of the loop to create another process to execute another command. When this program processes 5 commands, the 5 processes that execute the commands will be created to run at the same time. There is concurrency among the parent and all of the child processes.
The essential difference between the two programs (other than using different operating systems) is the way the execution of the 6 processes is coordinated. In Figure 8.1(a), the parent and child synchronize when the child terminates. In Figure 8.1(b), there is no synchronization between the parent and child processes.

Here is another example from Chapter 2: In the Laboratory Exercise at the end of the chapter, the challenge is to write a set of concurrent programs so that a parent thread creates N child threads, then signals the child threads when they are supposed to halt. (Observe that this is synchronization among a set of threads in a process, instead of among a set of classic processes.) In this case, the child threads periodically attempt to synchronize with the parent. If the parent has not issued the signal to synchronize, then the child proceeds with its computation. Here is a much simpler version of the parent code than what appeared in the exercise (although it does almost the same thing):

```c
static int runFlag = TRUE;
void main(...)
{
    ...
    // For 1 to N
    for (i = 0; i < N; i++) {
        // Create a new thread to execute simulated work
        CreateThread(...);
    }
    // runtime is the number of seconds that the children should run
    // Sleep while children work ...
    Sleep(runtime*1000);
    RunFlag = FALSE;
    ...
}
```

The `Sleep(K)` call causes the thread to sleep for K milliseconds, meaning that `Sleep(1000)` would cause the thread to sleep for a second. This code skeleton uses system time to determine how long a child thread should run. While the child threads work for `runTime` seconds, the parent thread sleeps. Here is the code skeleton that each child thread executes:

```c
DWORD WINAPI threadWork(LPVOID threadNo) {
    ...
    while(runFlag) {
        // Do one iteration of work, then check the runFlag
    }
    // The parent just signaled me to halt
    return result;
}
```

Figure 8.2 is another graph depiction of the parent and child thread behaviors. In this graph model, there are N+1 threads executing concurrently. Instead of synchronizing when a child terminates, each child thread attempts to synchronize at the end of an iteration of its work. If it receives a halt signal from the parent, it terminates; otherwise, it does another iteration of work.
The parent thread creates \( N \) child threads, each running as an iterative loop. At the end of the loop, each child checks to see if the \( \text{runFlag} \) has been set \text{FALSE}. If it has not, then the child iterates through the loop again. If the \( \text{runFlag} \) has been set \text{FALSE}, then the child terminates.

This technique works for threads that share an address space, but will generally not work for processes that do not share an address space. As you will see later in this section, this simple technique for synchronizing threads can fail. Providing robust and useful synchronization mechanisms is a fundamental problem in multiprogramming operating systems. This chapter first explains why this can be such a difficult problem, and then how this problem can be addressed by incorporating a basic mechanism—the \textit{Dijkstra semaphore}—into the OS.

For concurrency to be useful, it must be possible to construct concurrent programs so that threads can share information, yet not interfere with one another during certain critical parts of their respective execution. The main barriers to effective use of concurrency in applications arise from the following issues:

- Software technologies have not converged on any set of generally applicable programming paradigms for concurrent programs. Each concurrent programming solution potentially requires a new approach and design. This is a primary focus of applied distributed programming.

- Synchronization is usually at the heart of any concurrent programming implementation. Operating systems classically provide only the minimum mechanism to support synchronization and concurrency, since there are so many different ways to
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:

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

Code schema for \( p_1 \)                               Code schema for \( p_2 \)
...                                                 ...
balance = balance+amount;                           balance = balance-amount;
...                                                 ...
```

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

```assembly
Code schema for \( p_1 \)                               Code schema for \( p_2 \)
load R1, balance                                      load R1, balance
load R2, amount                                       load R2, amount
add R1, R2                                           sub R1, R2
store R1, balance                                     store R1, balance
```

Now suppose \( p_1 \) is executing the machine instruction

```assembly
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.
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;
$p_1$ indicates that it has completed its critical section. Eventually, $p_1$ will again be interrupted by the timer, and then $p_2$ can finally enter its critical section and continue with its work.

**FIGURE 8.7  The Execution Pattern**
While $p_1$ is in its critical section, $p_2$ waits at its while statement. In this case, $p_2$ will use entire timeslices executing the wait code.

While the approach shown is conceptually sound, it introduces a new critical section related to testing and setting the lock variable. If a thread is interrupted immediately after executing the while statement, but before it sets the lock, then the solution fails: Both processes can be in their critical sections at the same time. The problem is that manipulating the lock variable is, itself, a critical section: You have to solve a small critical section problem (manipulating lock) before you can solve the original critical section problem (manipulating balance)!

There is an important difference between the critical section to manipulate lock, and the one to manipulate balance: The lock critical section will be exactly the same code every time a process wants to enter a critical section. But the balance critical section code is determined by the application, which means that it might take a long time to execute, or even contain an infinite loop. Using this knowledge about the lock manipulation algorithm, we recognize that it would generally be acceptable to disable interrupts while we test and set the lock variable, since it will only be 3–4 machine instructions. But since enableInterrupts() and disableInterrupts() are privileged, we can define two new OS system calls, enter() and exit(), as shown in Figure 8.8. With these routines, a process calls enter() when it wants to enter a critical section, then calls exit() when it leaves the critical section. In this case, interrupts are disabled only by the OS code (while lock is being manipulated). Even when a process is blocked, waiting to enter its critical section, the interrupts are only disabled for a few instructions at a time. This avoids the
problems caused by disabling the interrupts for an extended period of time since an inter-
rupt 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 avail-
able, 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.

```c
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 sec-
tion 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:

```c
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, sub-
tle 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 sec-
ond 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, manip-
ulating a common list. Each task is able to add or delete an entry, requiring that the list
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 incre-
mented. 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 \textit{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 \texttt{lock1}.
- Thus \( p_2 \) will be prevented from entering its critical section to manipulate the list
  when it tests \texttt{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 \texttt{<intermediate computation>} (after hav-
  ing set \texttt{lock1} to \texttt{TRUE}) and \( p_2 \) begins to execute.
- \( p_2 \) will set \texttt{lock2} and then wait for \texttt{lock1} at the while statement.
- Eventually, the clock interrupt will resume \( p_1 \), and \( p_1 \) can then complete the
  \texttt{<intermediate computation>} and block at its while statement test on \texttt{lock2}
  (prior to updating the descriptor).
\section*{Multiple Shared Variables with Disabled Interrupts}

This attempted solution uses \texttt{enter()} and \texttt{exit()} to encapsulate two critical sections. The problem is that \texttt{p1} could be interrupted after it deletes an element from the list but before it updates the header. If \texttt{p2} runs next, then the list and the header will be inconsistent.

\begin{verbatim}
shared boolean lock1 = FALSE; /* Shared variables */
shared boolean lock2 = FALSE;
shared list L;

Program for \texttt{p1}

... /* Enter crit section to */
/* delete elt from list */
enter(lock1);
<delete element>;
/* Exit critical section */
exit(lock1);

<intermediate computation>;
/* Enter crit section to */
/* update length */
enter(lock2);
<update length>;
/* Exit critical section */
exit(lock2);

... /* Enter crit section to */
/* add elt to list */
enter(lock2);
<add element>;
/* Exit critical section */
exit(lock2);

... /* Exit both crit sections */
exit(lock1);
exit(lock2);
... 
\end{verbatim}

\section*{Ensuring Consistency in Related Values}

In this attempted solution to the list manipulation problem, the two processes may deadlock if \texttt{p1} is interrupted after it acquires \texttt{lock1}, but before it acquires \texttt{lock2}, and \texttt{p2} updates the list.

\begin{verbatim}
shared boolean lock1 = FALSE; /* Shared variables */
shared boolean lock2 = FALSE;
shared list L;

Program for \texttt{p1}

... /* Enter crit section to */
/* delete elt from list */
enter(lock1);
<delete element>;
/* Exit critical section */
exit(lock1);

<intermediate computation>;
/* Enter crit section to */
/* update length */
enter(lock2);
<update length>;
/* Exit critical section */
exit(lock2);

... /* Enter crit section to */
/* add elt to list */
enter(lock2);
<add element>;
/* Exit critical section */
exit(lock2);

... /* Exit both crit sections */
exit(lock1);
exit(lock2);
... 
\end{verbatim}
However, now these two processes will deadlock: Neither thread can ever proceed, since each holds the lock that the other needs. This forms a deadlock between $p_1$ and $p_2$, where the locks are abstract resources. In our continuing exploration of synchronization approaches, we must guard against the possibility of a deadlock. Deadlock can also occur in any situation where multiple processes compete for resources (not just in the critical section case). We will study the general case in Chapter 10.

**Resource Sharing**

The software critical section discussed earlier existed because the two processes shared the `balance` variable. The solutions synchronized the two processes so that one or the other of them had access to the shared variable at any given time. The traffic example suggested that critical sections can occur with any kind of time-multiplexed, shared resource such as the intersection of the two streets.

The processes/threads in a concurrent application will necessarily share some resource, such as variables, but also files, buffers, and devices. For example, if an application is intended to manage a company’s inventory, one process may run threads that handle the delivery of products from the inventory. A second process might handle ordering of goods whenever the inventory gets low. This means that the delivery process and the order process will need to access common information that describes the current inventory (even if it is only a file).

Many of these resources will be time-multiplexed, reusable resources. When a process or thread gains control of the resource, its access is mutually exclusive to other processes or threads within a process, meaning that no other process or thread can use the resource while it is allocated to one process or one thread. You should think of shared variables as shared resources with mutually exclusive access. Then the critical section problem is a special case of the **mutual exclusion** problem, which is the problem of time-multiplexing a shared resource so that only one process or one thread can use the resource at a time. In the critical section variant, the abstract resource is the critical section: Processes are mutually excluded from entering their critical section at the same time.

Our approach to synchronization solutions (and problems) has relied on the concurrent processes being executed on a single machine. In the `enter()`/`exit()` solution, it must be possible to prevent certain processes from executing by disabling interrupts. In all the other solutions, we rely on the presence of **shared memory**, for example, to hold a `lock` variable that can be tested and set by the concurrent processes. As we saw in Chapter 6, process address spaces prevent memory sharing. If our strategy is to synchronize using shared variables, the OS must provide some mechanism to override address space barriers so independent processes can access the shared variable.

### 8.2 Evolving from the Classic Solution

The need for interactions among threads (possibly in different processes) leads to the need for synchronization, and synchronization introduces the critical section and deadlock problems. Recall that in Chapter 2, `FORK()`, `JOIN()`, and `QUIT()` were introduced as mechanisms for creating and destroying processes. They can also be used to synchronize
processes involved in a concurrent computation. In Figure 8.12(a), the schematic diagram represents how a computation can use \texttt{FORK()}, \texttt{JOIN()}, and \texttt{QUIT()} to achieve concurrency. The initial process (or one of its descendants) creates a process, A, by performing a \texttt{FORK()} operation. Process A then executes \texttt{FORK()} to create process B. A and B then \texttt{JOIN()}, leaving only one process running, say B. Next, process B executes a \texttt{FORK()} to create process C. Processes B and C execute concurrently and then \texttt{JOIN()}, leaving only one process, again say B. Finally, process B performs a \texttt{QUIT()} operation to complete the computation. In this example, there are three distinct processes in the application schema (ignoring the initial process).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig8-12.png}
\caption{Synchronization using \texttt{FORK()}, \texttt{JOIN()}, and \texttt{QUIT()}}
\end{figure}

\textbf{Part (a) represents a computation in which the concurrent tasks are executed by repeatedly creating and destroying processes. Part (b) is an alternative approach in which synchronization operators are used in lieu of some of the process creations and destructions.}

Figure 8.12(b) represents an alternative approach using a synchronization operator ("Synchronize" in the figure). Again, process (or thread) A is created, either directly or indirectly, by the initial process. Process A creates process B using a \texttt{FORK()} command. At some of the points where \texttt{FORK()} and \texttt{JOIN()} were used to create and destroy processes in Figure 8.12(a), the two processes explicitly synchronize their operation so that neither will proceed beyond a certain point in the computation until both have reached their respective synchronization points. In the analogies at the beginning of the chapter, this kind of synchronization is similar to that used in the espionage movie: Processes A and B wait until both have reached a certain point in their execution (or the clock reaches 5:00 in the espionage example). Neither process uses a \texttt{JOIN()} or \texttt{QUIT()}, so neither
process terminates. Instead, the first one to reach its synchronization point suspends itself until the other has reached its synchronization points. In the figure, process A then waits while B performs sequential processing. The processes then synchronize again and continue concurrent execution. After all the work is done, the two processes `JOIN()`. In Figure 8.12(b), process A is assumed to be the last to execute `JOIN()`, so it continues and finally performs a `QUIT()` to finish the computation. The processes in the schematic diagram could also be threads—in one process or in different processes.

The two approaches in Figure 8.12 perform the same computation with the same amount of overlapped operation. Which approach is preferable? Since the introduction of `FORK()`, `JOIN()`, and `QUIT()`, OS designers have observed that process creation and destruction tend to be quite costly operations because they require considerable manipulation of process descriptors, protection mechanisms, and memory management mechanisms. The synchronization operation, on the other hand, can be thought of as a resource request on a consumable resource, such as a shared Boolean variable, and can be implemented far more efficiently. Because the amount of time to create/destroy a process is three or more orders of magnitude larger than for synchronization, the trend in contemporary operating systems is to use synchronization mechanisms to complement the process/thread creation/destruction operations.

In Section 8.1, some basic ideas were discussed that showed how synchronization might be accomplished using locks on a shared variable. The example solution relied on the semantics of the particular problem. In a generalized mechanism, a thread should be able to block until some previously defined event has occurred in another thread (possibly in another process). Consider the example introduced in Figure 1.8 and repeated in Figure 8.13. The intent of these code segments is that `proc_B` should not execute its first `read()` statement until `proc_A` completes the `write()` for the variable `x`. Further, this synchronization should take place each time through the loop. When `proc_B` starts, it should suspend operation until the `write(x)` event occurs in `proc_A`. This is a variant of the critical section problem in that the synchronization is needed to ensure cooperation between the threads executing `proc_A` and `proc_B`, instead of to resolve competition for access to a critical section.

There are three basic approaches to implementing synchronization strategies:

- Use only user mode software algorithms and shared variables.
- Disable and enable interrupts around critical sections as indicated in Figure 8.5, although as already pointed out, such solutions potentially have a dramatic effect on the I/O system.
- Incorporate specialized mechanisms in the hardware and/or operating system to support synchronization. This approach was first proposed by Edsger Dijkstra [Dijkstra, 1968] and remains the basis of the today’s solutions. It relies on the use of the semaphore abstract data type implemented by the operating system.

Two of the three approaches involve explicit OS support. The semaphore approach (and its extensions) is much preferred to the software and interrupt-based solutions.
These code fragments represent the activity of two different concurrent processes. \texttt{proc\_A} writes a shared variable, \( x \), and reads a shared variable, \( y \). \texttt{proc\_B} reads \( x \) and writes \( y \).

shared double \( x \), \( y \); /* Shared variables */

\texttt{proc\_A()} {
  while(TRUE) {
    <compute A1>;
    write(x); /* Produce x */
    <compute A2>;
    read(y); /* Consume y */
  }
}

\texttt{proc\_B()} {
  while(TRUE) {
    read(x); /* Consume x */
    <compute B1>;
    write(y); /* Produce y */
    <compute B2>;
  }
}

\section*{8.3 \textbullet 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 \textit{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:

\begin{itemize}
  \item Only one process at a time should be allowed to be executing in its critical section (mutual exclusion).
  \item 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).
  \item 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.
  \item 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.
\end{itemize}

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

\texttt{fork(proc, N, arg1, arg2, \ldots, argN)}

means that a single-threaded process is created and begins executing \texttt{proc()} in its own address space using the \( N \) arguments provided. \texttt{<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 \texttt{proc\_0()}, and the other executing \texttt{proc\_1()}.

```c
proc\_0() {
    while(TRUE) {
        <compute section>;
        <critical section>;
    }
}
<shared global declarations>;
<initial processing>;
fork(proc\_0, 0);
```

```c
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($s$): $[\text{while}(s == 0) \{\text{wait}\};\ s = s - 1]$

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, \texttt{mutex}, to be one. Each process invokes the \texttt{P} operation to enter the critical section, and the \texttt{V} operation when it leaves the critical section.

```c
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);
```

```c
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 \texttt{P} and \texttt{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 \texttt{busy} and \texttt{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.

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

```c
proc_B() { 
    while(TRUE) { 
        P(s1); /* Wait for proc_A signal */
        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.

```c
/* 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 set-
ting 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 infor-
mation (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 prob-
lem. 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 oper-
ation by blocking the producer when there are no empty buffers and blocking the con-
sumer when there are no full buffers.

The buffers are a block of memory logically split into N parts. Each buffer must con-
tain 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 pro-
tects 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 pro-
ducer if there are no empty buffers. Similarly, the P(full) operation blocks the consumer
if there are no full buffers.
The Bounded Buffer Problem

This solution uses three semaphores: \texttt{mutex} is a binary semaphore, while \texttt{full} and \texttt{empty} are general semaphores (taking on values from 0 to \(N\)). The \texttt{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.

\begin{verbatim}
producer() {
  bufType *next, *here;
  while(TRUE) {
    produceItem(next); /* 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, fullPool);
    V(mutex);
    /* Signal a full buffer */
    V(full);
  }
}
consumer() {
  bufType *next, *here;
  while(TRUE) {
    consumeItem(next);
    /* 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);
\end{verbatim}

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 \texttt{reader} process can share the resource with any other reader process, but not with any writer process. A \texttt{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.
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.

```c
reader() { 
    while(TRUE) { 
        <other computing>
        P(mutex);
        readCount = readCount+1; /* Critical section */
        if (readCount == 1) 
            P(writeBlock);
        V(mutex);
        /* Critical section */
        access(resource);
        P(mutex);
        readCount = readCount-1;
        if(readCount == 0) 
            V(writeBlock);
        V(mutex);
    }
}
writer() { 
    while(TRUE) { 
        <other computing>
        P(writeBlock);
        access(resource);
        V(writeBlock);
    }
}
```

```c
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 */
```

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.

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

**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.
**Implementing Semaphores.** Figure 8.8 showed how interrupts can be disabled to manipulate a lock variable, but not during an entire critical section (as was done in Figure 8.5). A semaphore implementation following this model disables interrupts, but only for a short period of time, and wholly within the P and V function (see Figure 8.22). In the figure, a C++ style class is used to emphasize that the semaphore is implemented as an abstract data type with private data structures and function implementations, but exporting a public interface. Since P and V are operating system functions, the implementation code assumes that user processes can have a reference to the OS semaphore object, meaning that the `P(s)` semaphore function call would be called by a code segment such as

```c
semaphore *s;
...
    s = sys_getSemaphore();
...
    s->P();
```

The interrupts are enabled most of the time that a thread is blocked on a semaphore; they are disabled only while the semaphore’s value is manipulated. This has two important effects:

- There is minimal effect on the I/O system.
- While a process holds a semaphore, it only prevents those threads from running that are competing for a relevant critical section. All other threads are unaffected by the fact that one thread is in its critical section.

Semaphores can be implemented without disabling interrupts if the hardware incorporates a few special provisions that the OS can use. Just as in the interrupt-based design, the OS can create an abstract resource for each semaphore. Then it uses resource managers such as those described in Section 6.7 to block processes when they perform a P operation just as if they had performed a request operation on a conventional resource. The issue becomes how the semaphore resource manager can correctly implement simultaneous access to the semaphore without using interrupts.

The **test-and-set (TS)** instruction is the dominant way to accomplish the effects of P and V in modern hardware. The hardware designer can add the TS instruction repertoire with little effort. By doing so, it can make semaphore implementation simple and efficient. The test-and-set instruction

```c
TS    R3, m // Test-and-set of location m
```

causes the contents of memory location `m` to be loaded into a register `R3` (with the condition code register for `R3` set to reflect the value of the data in `R3`) and memory location `m` to be rewritten with a value of **TRUE**. The essential aspect of **TS** is that it is a single machine instruction. Figure 8.23(a) shows the memory cell `m`, register `R3`, and the condition code for register `R3` before executing the TS instruction, and Figure 8.23(b) shows the result of executing the instruction.
FIGURE 8.22 Implementing Semaphores Using Interrupts

The P operation may cause the calling process to wait. In order to minimize the impact on the rest of the system, interrupts are enabled once per pass through the wait loop.

```cpp
class semaphore {
    int value;

public:
    semaphore(int v = 1) {
        // allocate space for the semaphore object in the OS
        value = v
    }

    P() {
        disableInterrupts();
        // Loop until value is positive
        while (value == 0) {
            enableInterrupts(); // Let interrupts occur
            disableInterrupts(); // Disable them again
        }
        value22;
        enableInterrupts();
    }

    V() {
        disableInterrupts();
        value--; // Decrease the value
        enableInterrupts();
    }
};
```

FIGURE 8.23 The Test-and-Set Instruction

The "TS R3, m" instruction loads register R3, tests its value, and writes a TRUE back to memory location m.

(a) Before Executing TS

(b) After Executing TS
Assume $TS$ is in the instruction repertoire of a machine and it is implemented as an OS function named $TS(m)$, where $m$ is a memory cell. Now the critical section problem can be solved as shown in Figure 8.24(a). (Part (b) is the corresponding code using $P$ and $V$, for comparison.) After the original value from location $s$ has been loaded, any interrupting process will detect the value stored in $s$ as being $TRUE$ and so will block at the while loop. This occurs even if an interrupt occurs before the process actually begins processing critical section code. The assignment statement’s resetting of $s$ is assumed to be atomic, since it also would normally be accomplished with a single machine instruction.

**FIGURE 8.24 Implementing the Binary Semaphore with $TS$**

The $P$ operation can be implemented by embedding the $TS$ instruction in a while loop that reads the condition code register. The $V$ operation can be implemented by a store instruction with an immediate operand.

```plaintext
boolean s = FALSE; semaphore s = 1;
...
while(TS(s)) ; P(s);
   <critical section>;
   <critical section>;
s = FALSE;
V(s);
...
(a)

(b)
```

One apparent shortcoming of $TS$ is that it replaces only the $P$ operation for binary semaphores—the ones taking on only the values of 0 and 1. The obvious question is: Can it be used to implement a general semaphore? Since a counting semaphore can take on any nonnegative value, the $TRUE$ and $FALSE$ values of a binary semaphore cannot represent integers larger than 1. Figure 8.25 is an algorithm that implements general semaphores.

**FIGURE 8.25 Implementing the General Semaphore with $TS$**

This algorithm demonstrates that you can use the $TS$ instruction to implement general semaphores. The idea is that you use a binary semaphore, $s.mutex$, to guard a critical section where integer arithmetic is used to implement a range of semaphore values.

```plaintext
struct semaphore {
    int value = <initial value>;
    boolean mutex = FALSE;
    boolean hold = TRUE;
};
shared struct semaphore s;
P(struct semaphore s) {
    while(TS(s.mutex)) ;
    s.value = s.value - 1;
    if(s.value < 0) {
        s.mutex = FALSE;
        while(TS(s.hold)) ;
    } else
        s.mutex = FALSE;
}
V(struct semaphore s) {
    while(TS(s.mutex)) ;
    s.value = s.value + 1;
    if(s.value <= 0) {
        while(!s.hold) ;
        s.hold = FALSE;
    }
    s.mutex = FALSE;
}
```

Chapter Eight  Basic Synchronization Principles
In this figure, \texttt{s.mutex} is used to implement mutual exclusion, while a thread manipulates \texttt{s.value} to represent the general semaphore value. The \texttt{s.hold} Boolean is used to stage threads blocked by the semaphore. Thus, any thread waiting for the semaphore will be waiting at the statement

\begin{verbatim}
while (TS(s.hold));
\end{verbatim}

in the P procedure. When \texttt{s.hold} returns a value of \texttt{FALSE} (the V operation will have set the value \texttt{FALSE} in those cases when it detected threads queued on semaphore \texttt{s}), the thread invoking P will block at the \texttt{TS} in the outer \texttt{while} loop. Also, note that a thread executing the P operation will release the critical section related to manipulating the \texttt{s.value} entry before it begins waiting on the \texttt{s.hold} variable.

One other statement in the solution merits careful consideration. In the V operation, the

\begin{verbatim}
while (!s.hold);
\end{verbatim}

is required. This is because a race condition can occur in which a thread that is blocked in the P procedure, yet the V procedure encounters \texttt{s.hold} as being \texttt{TRUE}. This situation can occur when consecutive V operations occur before any thread executes a P operation. Without the while statement, the result of one of the V operations could be lost.

**Busy-Waiting.** A busy-wait condition refers to a case where the code repeatedly executes a loop that tests a variable until the variable switches values—as is the case in the \texttt{while} loop in Figure 8.22. We first encountered the busy-wait condition when learning how software controls devices (see Section 4.4). Although the implementation using interrupts (Figure 8.22) or the \texttt{TS} instruction (Figure 8.24 and Figure 8.25) greatly reduce the amount of busy waiting compared to the technique used in Figure 8.5, it can still be quite wasteful of CPU time.

Suppose the implementation shown in Figure 8.25 were used in a multiprogrammed uniprocessor system. Then, whenever a process is scheduled to run and it blocks on a semaphore, it will repeatedly execute the

\begin{verbatim}
while(TS(s.hold));
\end{verbatim}

instruction until the timer interrupt invokes the scheduler to multiplex the process off the processor and another process onto the processor. When the blocked process obtains its next timeslice, it will resume this busy-wait if \texttt{s.hold} is still \texttt{TRUE}. The result is that the blocked process is slowing down some other process that would eventually execute a V operation and allow the first to proceed. The blocked process needs to indicate to the OS that it cannot do anything useful at the moment. This can be done by executing the equivalent of the \texttt{yield} instruction (from Chapter 7). Each time the process detects it is blocked, it might yield to another process that can perform useful work. This method suggests that the busy-waiting statement should be changed to

\begin{verbatim}
while (TS(s.hold))
    yield (*, scheduler);
\end{verbatim}

to eliminate wasting the unused portion of the timeslice by a blocked process.

**Active and Passive Semaphore Implementations.** A semaphore value is a consumable resource. A process/thread blocks if it requests a positive semaphore value but the semaphore is zero. When a process encounters a zero-valued semaphore, it moves
from the running state to the blocked state. From this perspective, the P operation is a resource request operation. A process moves from blocked to running when it detects a positive value of the semaphore; it decrements the semaphore at the same time it changes state. When another process releases a resource, by performing a V operation, the resource allocator moves the first process from blocked to ready. Being in the ready state does not mean the process is physically executing on the CPU, but it is at least on the ready list. This style of operation creates the possibility for another implementation complexity: If one process performs a V operation, should the OS “guarantee” that a waiting process will immediately perceive the action?

Figure 8.15 described the account credit and debit processes using a semaphore, mutex, to synchronize their access to the balance variable. Suppose proc_0 obtains the semaphore and enters the critical section, while proc_1 blocks on its P(mutex) operation. Suppose further that proc_0 exits the critical section and then executes V(mutex), and its own P(mutex)—all prior to proc_1 actually having an opportunity to detect that the semaphore took on a positive value. Then proc_1 could be prevented from entering its critical section even though the semaphore took on positive values while proc_1 was waiting for the semaphore.

This scenario is most likely to occur on a multiprogrammed uniprocessor if proc_0 does not yield the CPU immediately after incrementing the semaphore. In implementing the V operation, one is advised to add a yield() to the procedure definition immediately after incrementing the semaphore. This form of implementation is called the active V operation. This contrasts with the passive V operation, where the implementation increments the semaphore with no explicit context switch.

There is another aspect to semaphores that is highlighted with active and passive semaphores. Programmers sometimes treat the P operation as a “wait-for-event-occurrence” operation and the V operation as a “signal” to the waiting process (as in Figure 8.16). If the waiting process (the process blocked on a P operation) is not allowed to run at the time the signal is raised, should the event signaled by the V operation still be considered to be TRUE at the time the P operation finally sees the signal? We will revisit this issue in Chapter 9, where monitors are discussed.

8.4 • Synchronization in Shared Memory Multiprocessors

Sections 8.1 and 8.3 described a technique for implementing semaphores by disabling interrupts. In a shared-memory multiprocessor, this won’t work because disabling the interrupts on one CPU does not affect the interrupts on other CPUs. Therefore, commercial shared-memory multiprocessors all use specialized instructions such as TS to implement semaphores.

When a thread uses the busy-wait technique (without using yield), the only CPU unable to process other work is the one on which the blocked thread executes. Another thread capable of executing the V operation can be running on a different CPU. Therefore, the busy-wait is a mechanism for very fast recognition of the instant at which a thread becomes unblocked—the busy-wait bug becomes a busy-wait feature. In some cases, it is worth using one of N processors to poll the s.hold variable in order to detect the earliest possible moment the blocked thread becomes unblocked.
Operating systems for shared-memory multiprocessor systems typically support this scenario by including spin locks in the system call interface. A spin lock is a procedure that repeatedly performs the TS instruction to test a specified lock variable. To complete the abstract machine interface to the lock, there will be calls to create the lock, to lock and unlock it, and to block on the lock, and often a nonblocking call on the lock. This latter call is used so that if a thread detects it is locked out of a critical section, it can do other operations.

### 8.5 • Summary

Concurrent applications are made up of a group of processes/threads that share some resources used to solve the common problem. This introduces the critical section problem, which in turn makes it possible for two or more processes to become deadlocked. You saw several ways that you could address these problems: You could disable interrupts while code is in a critical section, or better, only while the OS manipulates lock variables. You could also use the classic `fork()`, `join()`, and `quit()` system calls, although those tend to be too slow to be effective. This sets the stage for Dijkstra semaphores.

Semaphores are the basic mechanism underlying all contemporary OS synchronization mechanisms. From a pragmatic viewpoint, semaphores can be implemented in the OS if the hardware supports the TS instruction. The TS instruction can be used to implement binary semaphores directly, or as a component in software to implement general semaphores. The straightforward TS implementation of a semaphore leads to busy-waiting. This waiting can be addressed by constructing the semaphore implementation so that it interacts with the scheduler. When a thread enters a phase of busy-waiting, it should yield to the scheduler. Finally, you saw that there can be active and passive implementations of semaphores. An active implementation calls the scheduler each time the semaphore value changes, but the passive implementation only calls the scheduler when a process blocks on the semaphore.

This chapter set the foundation of synchronization and discussed the complexity of dealing with synchronization using semaphores. The next chapter looks at synchronization abstractions.

### 8.6 • Exercises

1. Suppose processes $p_0$ and $p_1$ share variable $V_2$, processes $p_1$ and $p_2$ share variable $V_0$, and processes $p_2$ and $p_3$ share variable $V_1$.
   a. Show how the processes can use `enableInterrupt()` and `disableInterrupt()` to coordinate access to $V_0$, $V_1$, and $V_2$ so that the critical section problem does not occur.
   b. Show how the processes can use semaphores to coordinate access to $V_0$, $V_1$, and $V_2$ so that the critical section problem does not occur.

2. Enabling and disabling interrupts to prevent timer interrupts from invoking the scheduler is one way to implement semaphores. This technique can influence I/O because it makes the interrupt handler wait until the interrupts become enabled.
before the handler can complete an I/O operation. Explain how this could affect the accuracy of the system clock.

3. The following solution is alleged to be a solution to the critical section problem. Argue for its correctness or show a case in which it fails.

```c
shared int turn;
shared boolean flag[2];
proc(int i) {
    while (TRUE) {
        compute;
        /* Attempt to enter the critical section */
        try: flag[i] = TRUE; /* An atomic operation */
        while (flag[(i+1) mod 2]){ /* An atomic operation */
            if (turn == i) continue;
            flag[i] = false;
            while (turn != i);
            goto try;
        }
        /* Okay to enter the critical section */
        <critical section>;
        /* Leaving critical section */
        turn = (i+1) mod 2;
        flag[i] = false;
    }
}
turn = 0; /* Process 0 wins a tie for the first turn */
flag[0] = flag[1] = false;
/* Initialize flags before starting */
fork(proc, 1, 0); /* Create a process to run proc(0) */
fork(proc, 1, 1); /* Create a process to run proc(1) */
```

4. Dijkstra posed each of the following solutions as a potential software solution to the critical section problem and then explained why they failed [Dijkstra, 1968]. Provide your explanation about why they failed.

```
a. proc(int i) {
    while (TRUE) {
        compute;
        while (turn != i);
        critical_section;
        turn = (i+1) mod 2;
    }
}
shared int turn;
turn = 1;
fork(proc, 1, 0);
fork(proc, 1, 1);
```
b. proc(int i) {
    while (TRUE) {
        compute;
        while (flag[(i+1) mod 2]);
        flag[i] = TRUE;
        critical_section;
        flag[i] = FALSE;
    }
}

shared boolean flag[2];
flag[0] = flag[1] = FALSE;
fork(proc, 1, 0);
fork(proc, 1, 1);

c. proc(int i) {
    while (TRUE) {
        compute;
        flag[i] = TRUE;
        while (flag[(i+1) mod 2]);
        critical_section;
        flag[i] = FALSE;
    }
}

shared boolean flag[2];
flag[0] = flag[1] = FALSE;
fork(proc, 1, 0);
fork(proc, 1, 1);

5. In the solution to the bounded buffer problem (Figure 8.18), consider the ordering of the first two P operations in the producer and the consumer. Suppose the order of the P(full) and the P(mutex) instructions were reversed in the consumer. Would this solution still be correct?

6. Assume the writePending semaphore was omitted from Figure 8.21. Describe a simple sequence of reader and writer activity that causes the solution to fail for the second readers-writers policy.

7. Two processes, $p_1$ and $p_2$, have been designed so that $p_2$ prints a byte stream produced by $p_1$. Write a skeleton for the procedures executed by $p_1$ and $p_2$ to illustrate how they synchronize with one another using P and V.

8. The following is alleged to be a solution to the critical section problem. Argue for its correctness or show a case in which it fails.

```c
shared int turn; /* shared variable to synchronize operation */
boolean flag[2]; /* shared variable to synchronize operation */
```
proc(int i){
    while (TRUE) {
        <compute>
        flag[i] = TRUE; /* Attempt to enter the critical section */
        turn = (i+1) mod 2;
        while (((flag[(i+1) mod 2]) && (turn == (i+1) mod 2)));
        /* Now authorized to enter the critical section */
        <critical_section>;
        /* Exiting the critical section */
        flag[i] = FALSE;
    }
}

turn = 0;
flag[0] = flag[1] = FALSE;
fork(proc, 1, 0); /* Start a process on proc(0) */
fork(proc, 1, 1); /* Start a process on proc(1) */

9. In Chapters 4 and 5, you learned how device driver software synchronizes its behavior with device controller hardware (using the busy and done flags in the controller’s status register). In the generic schema shown in Figure 5.6, the driver starts the device in operation, writes the I/O details to the device state table, then halts. The device handler software reads the details from the device status table, completes the I/O operation, then returns from the system call (to the calling program). Some operating systems (such as Linux) use a slightly different approach, relying on the presence of a synchronization mechanism in the kernel. Instead of writing the status to a device status table and halting, the driver simply blocks until the device handler tells it to unblock and return to the caller. Write a pseudocode description for the device driver and handler that illustrates how this works.

10. The Sleepy Barber Problem [Dijkstra, 1968]. A barbershop is designed so that there is a private room that contains the barber chair and an adjoining waiting room with a sliding door that contains N chairs (see Figure 8.26). If the barber is busy, the door to the private room is closed and arriving customers sit in one of the available chairs. If a customer enters the shop and all chairs are occupied, the customer leaves the shop without a haircut. If there are no customers to be served, the barber goes to sleep in the barber chair with the door to the waiting room open. If the barber is asleep, the customer wakes the barber and obtains a haircut. Write code fragments to define synchronization schemes for the customers and the barber.
11. Provide a scenario in which a process executing the V procedure in Figure 8.25 will detect when \( s.value \) is less than or equal to 0 and then \( s.hold \) is TRUE.

12. Suppose a machine’s instruction set includes an instruction named \texttt{swap} that operates as follows (as an indivisible instruction):

\[
\texttt{swap(boolean *a, boolean *b)}
\]

\[
\begin{array}{l}
\text{\{ } \\
\text{ boolean t;} \\
\text{ t = *a;} \\
\text{ *a = *b;} \\
\text{ *b = t;} \\
\text{\} }
\end{array}
\]

Show how \texttt{swap} can be used to implement the P and V operations.

13. Semaphores are not implemented in older versions of UNIX, but processes with \texttt{stdout} of one process directed into \texttt{stdin} must synchronize their operation in a manner similar to that required in Problem 7. Write a program, \texttt{Source}, that copies a file to \texttt{stdout} and another program, \texttt{Sink}, that reads \texttt{stdin} and counts the number of bytes in the stream. Run \texttt{Source} and \texttt{Sink} with the output from \texttt{Source} piped into the input of \texttt{Sink}. How are the processes synchronized in your software?
Bounded Buffer Problem

This assignment can be solved using any Windows or POSIX-compliant OS.

The bounded buffer (producer-consumer) problem is a classic synchronization problem introduced by Dijkstra to illustrate two different ways to use his semaphores (see Section 8.3). In this exercise, you will design two threads to execute in a single address space. A producer thread creates “widgets,” and places each widget in an empty buffer for consumption by the consumer thread. The consumer retrieves the widget from the buffer, then releases the buffer to an empty buffer pool. If there are no full buffers, the consumer is blocked until new widgets are produced. If there are no empty buffers available when the producer creates a widget, the producer thread must wait until the consumer thread releases an empty buffer.

This problem asks you to design and implement a process with a producer and consumer thread using \( N \) different buffers (use a fixed size for \( N \) of 25). Base your solution on the solution to the producer-consumer problem shown in semaphore examples in Section 8.3. You will need a mutual exclusion semaphore to prevent the producer and consumer from manipulating the list of buffers at the same time, a semaphore so that the producer can signal the consumer to start processing when it creates a full buffer, and another semaphore for the consumer to signal the producer when it creates an empty buffer.

There are three subsections in the Background section for this Laboratory Exercise: The first provides general information about the problem, the second describes Windows semaphores as they are used for threads in a single process, and the third describes POSIX threads and their synchronization mechanisms. Read the first subsection and the subsection for the system you will use to solve the problem. The other subsection is optional reading.

Background

Threads in a single process all use the same address space and resources to solve a common problem. Since they are sharing resources, they usually need to coordinate their execution so that they do not interfere with one another. The Windows synchronization mechanisms all work with threads in a process. For UNIX, you will use the POSIX thread (or pthread) package. Most (but not all) UNIX systems support pthreads.

The following pseudocode is adapted from Figure 8.18 (a solution provided in an example in the chapter). As discussed in the main part of the chapter, a producer and a consumer thread are created by a parent thread (all in one modern process). The parent controls the length of time that the child threads run. Meanwhile, they continuously produce and consume widgets in buffers.
int runFlag = TRUE;
// pointers to semaphores (you will define the semaphore type
// These are globals and shared
semaphore empty;
semaphore full;
semaphore bufManip;
struct buffer_t {
  int buffer[N];
  unsigned int nextFull;
  unsigned int nextEmpty;
} widgets;

// The main program establishes the shared information used by
// the producer and consumer threads
main() {
  // Local variables
  int runTime; // Amount of time to execute
  int i;

  // Get a value for runTime
...
// Initialize synchronization objects
  empty = create_sync_object(N);
  full = create_sync_object(0);
  bufManip = create_sync_object(1);
// Initialize buffer pool
  widgets.nextEmpty = 0;
  widgets.nextFull = 0;
  for(i = 0; i < N; i++)
    widgets.buffer[i] = EMPTY;

// Create producer and consumer threads
  create_child_thread(&prod_thrd, NULL, producer, &widgets);
  create_child_thread(&cons_thrd, NULL, consumer, &widgets);
// Sleep while the children work ...
  sleep(runTime);
  runFlag = FALSE; // Signal children to terminate

// Wait for producer & consumer to terminate
...

// Release the semaphores
  delete_sync_object(empty);
  delete_sync_object(full);
  delete_sync_object(bufManip);

// Now we can quit
  printf("Main thread: Terminated\n");

  exit(1);
}
... producer(void *wp) {
    struct buffer_t *widgPtr;
    widgPtr = (struct buffer_t *) wp; // Cast buffer pointer
    srand(P_RAND_SEED); // Set random# seed
    itCount = 100;
    while(runFlag) {
        // Produce the buffer
        usleep(rand()%timeToProduce); // Simulate production time
        // Get an empty buffer
        P(empty);
        // Manipulate the buffer pool
        P(bufManip);
        widgPtr->buffer[widgPtr->nextEmpty] = itCount++;
        widgPtr->nextEmpty = (widgPtr->nextEmpty+1) % N;
        V(bufManip);
        V(full);
    }
    // Terminate
    ...;
}

... *consumer(void *wp) {
    struct buffer_t *widgPtr;
    widgPtr = (struct buffer_t *) wp;
    srand(C_RAND_SEED); // Set random# seed
    runFlag = TRUE;
    while(runFlag) {
        // Get a full buffer
        P(full);
        // Manipulate shared data structure
        P(bufManip);
        itCount = widgPtr->buffer[widgPtr->nextFull];
        widgPtr->nextFull = (widgPtr->nextFull+1) % N;
        V(bufManip);
        // Consume the buffer
        usleep(rand()%timeToConsume); // Simulate consumption
        V(empty);
    }
    // Terminate
    ...;
}

Your job is to define all the italicized functions for thread management and synchronization. Feel free to use this code segment. Substitute appropriate function names for your target platform (or use these function names as wrappers if you wish). Next, we will look at the OS-specific background information (first Windows, then POSIX threads).
Synchronizing Threads in Windows

There are several different synchronizing mechanisms in Windows, including the Mutex and Semaphore. A thread synchronizes on a Mutex, Semaphore, or other OS synchronization object using a wait function. That is, a wait function is similar to a Dijkstra P operation in that a thread calls it whenever it wants to obtain a semaphore or enter a critical section. When a thread calls a wait function, the thread is blocked until the internal state of the synchronization object determines that it is okay for the calling thread to proceed. The most commonly used wait function is WaitForSingleObject():

```c
DWORD WaitForSingleObject(
    HANDLE hHandle; // handle of object to wait for
    DWORD dwMilliseconds; // time-out interval in
    // milliseconds
);
```

The hHandle parameter is the handle for the synchronization object. The dwMilliseconds parameter specifies a maximum amount of time (in milliseconds) that the thread is willing to wait for the object to complete the synchronization. You can use GetLastError() to see if the function returned because either a notification was sent by the synchronization object (GetLastError() returns WAIT_OBJECT_0) or the maximum amount of time expired (WAIT_TIMEOUT). You can also use a value of INFINITE for dwMilliseconds, which means that the calling thread will block until it receives a notification from the object and will never time out. WaitForSingleObject() can be used with Mutex and Semaphore objects. (WaitForSingleObject() infers the type of object it is waiting for from the object handle.)

**Mutex objects** are especially built to handle the critical section problem. A Mutex object may have an owner thread, or be unowned. Having ownership of the object means that the thread is “holding the mutex.” A thread can become the owner of a Mutex object when the object is created, when a handle to it is opened, or by a wait function. To understand the details, consider the function prototypes:

```c
HANDLE CreateMutex(
    LPSECURITY_ATTRIBUTES lpMutexAttributes, // pointer to security attributes
    BOOL bInitialOwner, // flag for initial ownership
    LPCTSTR lpName // pointer to mutex-object name
);
```

The bInitialOwner attribute determines whether or not the calling thread will be the owner of the Mutex object. If bInitialOwner is set to TRUE (and the function call succeeds), the Mutex object will be created in a state that can be obtained by another thread. CreateMutex() can fail if it selects a name that is already in use—GetLastError() will return the value ERROR_ALREADY_EXISTS.

Once the Mutex object is created, any thread in the calling thread’s process can use it. If threads in other processes intend to use the Mutex object, then they must know the name of the Mutex object and use OpenMutex() with the correct name.

If a thread wishes to hold a Mutex object, it uses a wait function to request control of the object. A successful call on a Mutex object (you must check the return code if you
allow a timeout return) causes the calling thread to gain control of the Mutex. The 
\texttt{ReleaseMutex()} function call releases the Mutex.

Mutex objects can be used to solve the critical section problem. Suppose that threads 
X and Y share resource R—both threads perform some computation, access R, then per-
form more computation. Since R is a shared resource, the access is a critical section. Here 
is a code skeleton to handle the problem using a Mutex object:

```c
int main(...) {
    ...
    // Open resource R
    ...
    // Create the Mutex objects with no owner (signaled)
    mutexR = CreateMutex(NULL, FALSE, NULL);
    ...
    CreateThread(..., workerThrd, ...) ...; // Create thread X
    ...
    CreateThread(..., workerThrd, ...) ...; // Create thread Y
    ...
}
```

```c
DWORD WINAPI workerThrd(LPVOID) {
    ...
    while(...) {
        // Perform work
        ...
        // Obtain mutex
        while(WaitForSingleObject(mutexR) != WAIT_OBJECT_0);
        // Access the resource R
        ReleaseMutex(mutexR);
    }
    ...
}
```

Semaphore objects implement Dijkstra general semaphore semantics. That is, 
Semaphore objects are able to maintain a count to represent integer values (rather than the 
implicit binary values of Mutexes). A Semaphore object is created with a call to:

```c
HANDLE CreateSemaphore(
    LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
    // pointer to security attributes
    LONG lInitialCount, // initial count
    LONG lMaximmCount, // maximum count
    LPCTSTR lpName // pointer to semaphore-object name
);
```

A Semaphore object keeps an internal variable with a value between zero and 
lMaximmCount (which must be greater than zero). When the object is created, the ini-
tial value of the internal variable can be set to any value in the allowable range, and is 
specified by the lInitialCount argument. The state of the Semaphore object is deter-
mained by the value of the internal variable: If it is set to zero, a process that calls a wait
function on the Semaphore will block. If the Semaphore has a value in the range 
[1:1MaximumCount], a wait call will decrement the count and return.

The internal values of the Semaphore object are manipulated indirectly using func-
tions. The ReleaseSemaphore() function increases the internal variable count

BOOL ReleaseSemaphore(
    HANDLE hSemaphore,       // handle of the semaphore object
    LONG lReleaseCount,      // amount to add to the current count
    LPLONG lPreviousCount    // address of previous count
);

The lReleaseCount parameter specifies the amount to add to the semaphore (poten-
tially causing a state change in the object). The lPreviousCount is a pointer to a vari-
able that will be set to show the value of the count before ReleaseSemaphore() was
called (and may be set to NULL if you do not care about the previous value).

Semaphore objects are used in situations where you need to have the synchronization
mechanism count values. Suppose that threads X and Y are both using units of resource
R—either may request K units, use them for some period of time, then return them. Here
is a code skeleton to handle the problem using Semaphore objects:

#define N ...

int main(...) {
    // This is a controlling thread
    ...
    // Create the Semaphore object
    semaphoreR = CreateSemaphore(NULL, 0, N, NULL);
    ...
    CreateThread(..., workerThrd, ...) ...; // Create thread X
    ...
    CreateThread(..., workerThrd, ...) ...; // Create thread Y
    ...
}

DWORD WINAPI workerThrd(LPVOID) {
    While(...) {
        // Perform some work
        ...
        // Acquire K units of the resource
        for(i = 0; i < K; i++)
            while(WaitForSingleObject(semaphoreR) != WAIT_OBJECT_0);
        // Perform some work
        ...
        // Release the K units
        ReleaseSemaphore(semaphoreR, K, NULL);
        ...
    }
}
POSIX Threads

The POSIX thread package provides a comprehensive set of functions for creating, deleting, and synchronizing threads within a modern process. Some implementations are accomplished completely in user space (meaning that the OS implements classic threads, and a library exports the thread functions). Other implementations provide thread support in the OS, then use the pthread API to export the functions to application programs. In this exercise, it will not matter whether the pthread implementation is done with user space or OS threads, since you will call the functions on the API without having to know their implementation. There are many online reference materials for the pthread package, including man pages on UNIX systems that support pthreads. You will also find it helpful to use your favorite search engine to look for “pthread reference” on the Web.

Threads are created with the `pthread_create()` function. Here is the function prototype:

```c
int pthread_create(pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine)(void *),
    void *arg);
```

You will need to include the `pthread.h` header file to define various thread types (such as `pthread_t` and `pthread_attr_t`) and the function prototypes exported by the pthread package. This function creates a new thread to run in the address space of the calling thread. The `thread` argument is a pointer to a `pthread_t`, which is a reference to the pthread descriptor for the new thread. It is possible to define various attributes for the new thread (such as the stack size) using the `attr` argument. For this assignment you can use the default attributes, by using `NULL` for the `attr` argument. The third and fourth parameters specify the entry point for a function, `start_routine`, and its argument, `arg`. The return value is 0 on success, and nonzero on failure.

A parent thread can wait for a child thread to terminate (that is, synchronize with its termination) using this function:

```c
int pthread_join(pthread_t thread, void **status);
```

The `thread` argument is the `pthread_t` of the child thread. If the child returns a value using `pthread_exit(void *)`, it will be returned to the parent via the `status` argument. The child’s resources (such as its thread descriptor) will not be deallocated until the parent calls `pthread_join()`.

Suppose we wanted to create a new thread that would begin executing a function with a prototype

```c
void *worker_thread(void *a_list);
```

then we could accomplish that with the following code fragment:

```c
int main () {
   pthread_t my_thread;
   struct my_struct_t *my_struct;
   int *ret_val;
   ...
   my_struct = ...
}
// Define the argument list for the worker thread
if(!create(&my_thread, NULL, worker_thread, a_list)) {
    fprintf(stderr, ...);
    ...
}
...

// Wait for the child to terminate
if(!pthread_join(my_thread, &ret_val)) { // error return}
}

void *worker_thread(void *arg) {
    int *ret_val;
    ...
    // Work is completed, terminate
    pthread_exit(ret_val);
}

There are various synchronization primitives in the pthread API, including mutex, condition variable, and read/write lock. The condition variable primitive combines a mutex and another synchronization primitive to export a specialized mechanism that is not quite the right tool for this exercise. Condition variables were intended to be used in a more abstract mechanism called monitors, which are described in Chapter 9. The read/write lock is intended to be used for problems like the readers-writer problems described in the examples in Section 8.3. That is, multiple readers can simultaneously obtain an rwlock, but only one write can obtain it at a time. That leaves you with the mutex as the primitive to use for solving this Laboratory Exercise.

The pthread_mutex_t type is a synchronization primitive that is created as an unowned resource, and which threads can obtain ownership through the pthread_mutex_lock() call. They relinquish ownership with the pthread_mutex_unlock() call. So pthread_mutex_lock() is used to obtain mutually exclusive control of a critical section (like the Dijkstra P operation), and pthread_mutex_unlock() is used to release that control (like the V operation). Here are the function prototypes for creating a mutex (pthread_mutex_init()), for destroying a mutex (pthread_mutex_destroy()) and the two functions for manipulating a mutex:

int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);

The mutex argument in all of these functions is the reference to the pthread_mutex_t mutex descriptor. Just as you can provide attributes for a thread when you create it, you can also specify attributes for a mutex. However, the only attribute relates to performance behavior of the implementations, which we will not discuss in this book. Hence, you will use a value of NULL for the attr argument for pthread_mutex_init().

You can consult the documentation to find a few other functions to manipulate a pthread_mutex_t, although these are sufficient to solve the Laboratory Exercise.
Here is a code fragment to solve the account balance problem from Section 8.1 (also see Figure 8.15) using the `pthread_mutex_t`:

```c

pthread_mutex_t bal_mutex; // Global, accessible to all threads

int main () {
    ...
    pthread_mutex_init(&bal_mutex, NULL);
    ...
    // Create acct manager threads
    ...
}

void *acct_mgr(void *foo) { // foo is not used in this example
    ...
    amount = get_amount();
    switch(transaction) {
    case CREDIT:
        pthread_mutex_lock(bal_mutex);
        balance = balance + (double) *amount;
        pthread_mutex_unlock(bal_mutex);
        break;
    case DEBIT:
        pthread_mutex_lock(bal_mutex);
        balance = balance - (double) *amount;
        pthread_mutex_unlock(bal_mutex);
        break;
    ...
    }
}

Notice that `pthread_mutex_t` is a binary semaphore, but you will need a general semaphore to solve the problem. Fortunately, in the main part of this chapter, you learned an algorithm for implementing a general semaphore using only the test-and-set instruction. This will be a good place for you to get started thinking about how to implement a general semaphore for your solution.

● Attacking the Problem ●

Before you begin your solution, be sure to read the online documentation for the relevant function calls (the MSDN reference or UNIX `man` pages). Start with the pseudocode solution in the Background section. You may have to reformulate parts of the pseudocode, and you will certainly have to flesh out the entire implementation.