6.2 The Cigarette-Smokers Problem. Consider a system with three smoker processes and one agent process. Each smoker continuously rolls a cigarette and then smokes it. But to roll and smoke a cigarette, the smoker needs three ingredients: tobacco, paper, and matches. One of the smoker processes has paper, another has tobacco, and the third has matches. The agent has an infinite supply of all three materials. The agent places two of the ingredients on the table. The smoker who has the remaining ingredient then makes and smokes a cigarette, signaling the agent on completion. The agent then puts out another two of the three ingredients, and the cycle repeats. Write a program to synchronize the agent and the smokers using Java synchronization.

6.3 Explain why Solaris, Windows XP, and Linux implement multiple locking mechanisms. Describe the circumstances under which they use spinlocks, mutexes, semaphores, adaptive mutexes, and condition variables. In each case, explain why the mechanism is needed.

6.4 Describe how volatile, nonvolatile, and stable storage differ in cost.

6.5 Explain the purpose of the checkpoint mechanism. How often should checkpoints be performed? Describe how the frequency of checkpoints affects:

- System performance when no failure occurs
- The time it takes to recover from a system crash
- The time it takes to recover from a disk crash

6.6 Explain the concept of transaction atomicity.

6.7 Show that some schedules are possible under the two-phase locking protocol but not possible under the timestamp protocol, and vice versa.

Exercises

6.8 Race conditions are possible in many computer systems. Consider a banking system with two functions: deposit(amount) and withdraw(amount). These two functions are passed the amount that is to be deposited or withdrawn from a bank account. Assume a shared bank account exists between a husband and wife and concurrently the husband calls the withdraw() function and the wife calls deposit(). Describe how a race condition is possible and what might be done to prevent the race condition from occurring.

6.9 The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, \( P_1 \) and \( P_2 \), share the following variables:

```c
bool flag[2]; /* initially false */
int turn;
```

```c
do {
    flag[1] = true;
    while (flag[1]) {
        if (turn == 1) {
            flag[1] = false;
        } else {
            turn = 1;
            flag[1] = true;
        }
    } // critical section

    turn = 1;
    flag[1] = false;
    // remainder section
} while (true);
```

Figure 6.20 The structure of process \( P_1 \) in Dekker's algorithm.

The structue of process \( P_1 (i == 0 or 1) \) is shown in Figure 6.25; the other process is \( P_0 (j == 1 or 0) \). Prove that the algorithm satisfies all three requirements for the critical-section problem.

6.10 The first known correct software solution to the critical-section problem for \( n \) processes with a lower bound on waiting of \( n - 1 \) turns was presented by Eisenberg and McGuire. The processes share the following variables:

```c
enum psstate {idle, want_in, in, cs};
psstate flag[n];
int turn;
```

All the elements of flag are initially idle; the initial value of turn is immaterial (between 0 and \( n - 1 \)). The structure of process \( P_1 \) is shown in Figure 6.25. Prove that the algorithm satisfies all three requirements for the critical-section problem.

6.11 What is the meaning of the term busy waiting? What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answer.

6.12 Explain why spinlocks are not appropriate for single-processor systems yet are often used in multiprocessor systems.

6.13 Explain why implementing synchronization primitives by disabling interrupts is not appropriate in a single-processor system if the synchronization primitives are to be used in user-level programs.

6.14 Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.
do {
    while (TRUE) {
        flag[i] = want.in;
        j = turn;
        while (j != i) {
            if (flag[j] == idle) {
                j = turn;
                else
                    j = (j + 1) % n;
            }
        }
        flag[i] = in.ca;
        j = 0;
        while (j < n && (j == i || flag[j] == in.ca))
            j++;
        if (j >= n && (turn == i || flag[turn] == idle))
            break;
    }
    // critical section
    j = (turn + 1) % n;
    while (flag[j] == idle)
        j = (j + 1) % n;
    turn = j;
    flag[i] = idle;
    // remainder section
    while (TRUE);
}

Figure 6.20 The structure of process P in Eisenberg and McGuire's algorithm.

6.15 Describe two kernel data structures in which race conditions are possible. Be sure to include a description of how a race condition can occur.

6.16 Describe how the Swap() instruction can be used to provide mutual exclusion that satisfies the bounded-waiting requirement.

6.17 Servers can be designed to limit the number of open connections. For example, a server may wish to have only N socket connections at any point in time. As soon as N connections are made, the server will not accept another incoming connection until an existing connection is released. Explain how semaphores can be used by a server to limit the number of concurrent connections.

6.18 Show that, if the wait() and signal() semaphore operations are not executed atomically, then mutual exclusion may be violated.

6.19 Windows Vista provides a new lightweight synchronization tool called slim reader–writer locks. Whereas most implementations of reader–writer locks favor either readers or writers, or perhaps order waiting threads using a FIFO policy, slim reader–writer locks favor neither readers nor writers, nor are waiting threads ordered in a FIFO queue. Explain the benefits of providing such a synchronization tool.

6.20 Show how to implement the wait() and signal() semaphore operations in multiprocessor environments using the TestAndSet() instruction. The solution should exhibit minimal busy waiting.

6.21 Exercise 4.17 requires the parent thread to wait for the child thread to finish its execution before printing out the computed values. If we let the parent thread access the Fibonacci numbers as soon as they have been computed by the child thread — rather than waiting for the child thread to terminate — explain what changes would be necessary to the solution for this exercise? Implement your modified solution.

6.22 Demonstrate that monitors and semaphores are equivalent insofar as they can be used to implement the same types of synchronization problems.

6.23 Write a bounded-buffer monitor in which the buffers (portions) are embedded within the monitor itself.

6.24 The strict mutual exclusion within a monitor makes the bounded-buffer monitor of Exercise 6.23 mainly suitable for small portions.
   a. Explain why this is true.
   b. Design a new scheme that is suitable for larger portions.


6.26 How does the signal() operation associated with monitors differ from the corresponding operation defined for semaphores?

6.27 Suppose the signal() statement can appear only as the last statement in a monitor procedure. Suggest how the implementation described in Section 6.7 can be simplified in this situation.

6.28 Consider a system consisting of processes P1, P2, ..., Pn, each of which has a unique priority number. Write a monitor that allocates three identical line printers to these processes, using the priority numbers for deciding the order of allocation.

6.29 A file is to be shared among different processes, each of which has a unique number. The file can be accessed simultaneously by several processes, subject to the following constraint: The sum of all unique numbers associated with all the processes currently accessing the file must be less than n. Write a monitor to coordinate access to the file.
6.30 When a signal is performed on a condition inside a monitor, the signaling process can either continue its execution or transfer control to the process that is signaled. How would the solution to the preceding exercise differ with these two different ways in which signaling can be performed?

6.31 Suppose we replace the wait() and signal() operations of monitors with a single construct wait(B), where B is a general Boolean expression that causes the process executing it to wait until B becomes true.
   a. Write a monitor using this scheme to implement the readers-writers problem.
   b. Explain why, in general, this construct cannot be implemented efficiently.
   c. What restrictions need to be put on the wait statement so that it can be implemented efficiently? (Hint: Restrict the generality of B; see Kessels [1977]).

6.32 Write a monitor that implements an alarm clock that enables a calling program to delay itself for a specified number of time units (ticks). You may assume the existence of a real hardware clock that ticks a procedure tick in your monitor at regular intervals.

6.33 Why do Solaris, Linux, and Windows XP use spinlocks as a synchronization mechanism only on multiprocessor systems and not on single-processor systems?

6.34 In log-based systems that provide support for transactions, updates to data items cannot be performed before the corresponding entries are logged. Why is this restriction necessary?

6.35 Show that the two-phase locking protocol ensures conflict serializability.

6.36 What are the implications of assigning a new timestamp to a transaction that is rolled back? How does the system process transactions that were issued after the rolled-back transaction but that have timestamps smaller than the new timestamp of the rolled-back transaction?

6.37 Assume that a finite number of resources of a single resource type must be managed. Processes may ask for a number of these resources and—once satisfied—will return them. As an example, many commercial software packages provide a given number of licenses, indicating the number of applications that may run concurrently. When the application is started, the license count is decremented. If the application is terminated, the license count is incremented. If all licenses are in use, requests to start the application are denied. Such requests will only be granted when an existing license holder terminates the application and a license is returned.

The following program segment is used to manage a finite number of instances of an available resource. The maximum number of resources and the number of available resources are declared as follows:

```c
#define MAX_RESOURCES 5
int available_resources = MAX_RESOURCES;
```

When a process wishes to obtain a number of resources, it invokes the `decrease.count()` function:

```c
/* decrease available.resources by count resources */
/* return 0 if sufficient resources available, */
/* otherwise return -1 */
int decrease.count(int count) {
    if (available.resources < count)
        return -1;
    else { 
        available.resources -= count;
        return 0;
    }
}
```

When a process wants to return a number of resources, it calls the `increase.count()` function:

```c
/* increase available.resources by count */
int increase.count(int count) {
    available.resources += count;
    return 0;
}
```

The preceding program segment produces a race condition. Do the following:

a. Identify the data involved in the race condition.

b. Identify the location (or locations) in the code where the race condition occurs.

c. Using a semaphore, fix the race condition. It is ok to modify the `decrease.count()` function so that the calling process is blocked until sufficient resources are available.

6.38 The `decrease.count()` function in the previous exercise currently returns 0 if sufficient resources are available and -1 otherwise. This leads to awkward programming for a process that wishes to obtain a number of resources:

```c
while (decrease.count(count) == -1); 
```

Rewrite the resource-manager code segment using a monitor and condition variables so that the `decrease.count()` function suspends the process until sufficient resources are available. This will allow a process to invoke `decrease.count()` by simply calling `decrease.count(count);`