interested in Erlang and Scala, and in further details about functional languages in general, are encouraged to consult the bibliography at the end of this chapter for additional references.

5.11 Summary

Given a collection of cooperating sequential processes that share data, mutual exclusion must be provided to ensure that a critical section of code is used by only one process or thread at a time. Typically, computer hardware provides several operations that ensure mutual exclusion. However, such hardware-based solutions are too complicated for most developers to use. Mutex locks and semaphores overcome this obstacle. Both tools can be used to solve various synchronization problems and can be implemented efficiently, especially if hardware support for atomic operations is available.

Various synchronization problems (such as the bounded-buffer problem, the readers–writers problem, and the dining-philosophers problem) are important mainly because they are examples of a large class of concurrency-control problems. These problems are used to test nearly every newly proposed synchronization scheme.

The operating system must provide the means to guard against timing errors, and several language constructs have been proposed to deal with these problems. Monitors provide a synchronization mechanism for sharing abstract data types. A condition variable provides a method by which a monitor function can block its execution until it is signaled to continue.

Operating systems also provide support for synchronization. For example, Windows, Linux, and Solaris provide mechanisms such as semaphores, mutex locks, spinlocks, and condition variables to control access to shared data. The Pthreads API provides support for mutex locks and semaphores, as well as condition variables.

Several alternative approaches focus on synchronization for multicore systems. One approach uses transactional memory, which may address synchronization issues using either software or hardware techniques. Another approach uses the compiler extensions offered by OpenMP. Finally, functional programming languages address synchronization issues by disallowing mutability.

Practice Exercises

5.1 In Section 5.4, we mentioned that disabling interrupts frequently can affect the system’s clock. Explain why this can occur and how such effects can be minimized.

5.2 Explain why Windows, Linux, and Solaris implement multiple locking mechanisms. Describe the circumstances under which they use spinlocks, mutex locks, semaphores, adaptive mutex locks, and condition variables. In each case, explain why the mechanism is needed.
5.3 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.

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

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

5.6 Illustrate how a binary semaphore can be used to implement mutual exclusion among \( n \) processes.

---

**Exercises**

5.7 Race conditions are possible in many computer systems. Consider a banking system that maintains an account balance with two functions: `deposit(amount)` and `withdraw(amount)`. These two functions are passed the amount that is to be deposited or withdrawn from the bank account balance. Assume that a husband and wife share a bank account. 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.

5.8 The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, \( P_0 \) and \( P_1 \), share the following variables:

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

The structure of process \( P_i \) (\( i = 0 \) or \( 1 \)) is shown in Figure 5.21. The other process is \( P_j \) (\( j = 1 \) or \( 0 \)). Prove that the algorithm satisfies all three requirements for the critical-section problem.

5.9 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 pstate {idle, want_in, in_cs};
pstate 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 \( P_i \) is shown in Figure 5.22. Prove that the algorithm satisfies all three requirements for the critical-section problem.

5.10 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.
do {
    flag[i] = true;
    while (flag[j]) {
        if (turn == j) {
            flag[i] = false;
            while (turn == j)
                ; /* do nothing */
            flag[i] = true;
        }
    }
}

/* critical section */

turn = j;
flag[i] = false;

/* remainder section */
} while (true);

Figure 5.21 The structure of process $P_i$ in Dekker's algorithm.

5.11 Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.

5.12 The Linux kernel has a policy that a process cannot hold a spinlock while attempting to acquire a semaphore. Explain why this policy is in place.

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

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

5.15 Consider how to implement a mutex lock using an atomic hardware instruction. Assume that the following structure defining the mutex lock is available:

    typedef struct {
        int available;
    } lock;

(available == 0) indicates that the lock is available, and a value of 1 indicates that the lock is unavailable. Using this struct, illustrate how the following functions can be implemented using the test_and_set() and compare_and_swap() instructions:

- void acquire(lock *mutex)
- void release(lock *mutex)

Be sure to include any initialization that may be necessary.
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_cs;
        j = 0;

        while ((j < n) && (j == i || flag[j] != in_cs))
            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 5.22 The structure of process P_i in Eisenberg and McGuire's algorithm.

5.16 The implementation of mutex locks provided in Section 5.5 suffers from busy waiting. Describe what changes would be necessary so that a process waiting to acquire a mutex lock would be blocked and placed into a waiting queue until the lock became available.

5.17 Assume that a system has multiple processing cores. For each of the following scenarios, describe which is a better locking mechanism—a spinlock or a mutex lock where waiting processes sleep while waiting for the lock to become available:

- The lock is to be held for a short duration.
- The lock is to be held for a long duration.
- A thread may be put to sleep while holding the lock.
Chapter 5  Process Synchronization

```c
#define MAX_PROCESSES 255
int number_of_processes = 0;

/* the implementation of fork() calls this function */
int allocate_process() {
    int new_pid;

    if (number_of_processes >= MAX_PROCESSES)
        return -1;
    else {
        /* allocate necessary process resources */
        ++number_of_processes;

        return new_pid;
    }
}

/* the implementation of exit() calls this function */
void release_process() {
    /* release process resources */
    --number_of_processes;
}
```

**Figure 5.23** Allocating and releasing processes.

5.18 Assume that a context switch takes $T$ time. Suggest an upper bound (in terms of $T$) for holding a spinlock. If the spinlock is held for any longer, a mutex lock (where waiting threads are put to sleep) is a better alternative.

5.19 A multithreaded web server wishes to keep track of the number of requests it services (known as *hits*). Consider the two following strategies to prevent a race condition on the variable `hits`. The first strategy is to use a basic mutex lock when updating `hits`:

```c
int hits;
mutex.lock hit.lock;

hit.lock.acquire();
hits++;
hit.lock.release();
```

A second strategy is to use an atomic integer:

```c
atomic_t hits;
atomic_inc(&hits);
```

Explain which of these two strategies is more efficient.

5.20 Consider the code example for allocating and releasing processes shown in Figure 5.23.
a. Identify the race condition(s).

b. Assume you have a mutex lock named `mutex` with the operations `acquire()` and `release()`. Indicate where the locking needs to be placed to prevent the race condition(s).

c. Could we replace the integer variable

```c
int numberOfProcesses = 0
```

with the atomic integer

```c
atomic_t numberOfProcesses = 0
```

to prevent the race condition(s)?

---

**5.21** 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.

**5.22** Windows Vista provides a 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.

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

**5.24** Exercise 4.26 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—what changes would be necessary to the solution for this exercise? Implement your modified solution.

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

**5.26** Design an algorithm for a bounded-buffer monitor in which the buffers (portions) are embedded within the monitor itself.

**5.27** The strict mutual exclusion within a monitor makes the bounded-buffer monitor of Exercise 5.26 mainly suitable for small portions.

a. Explain why this is true.

b. Design a new scheme that is suitable for larger portions.

**5.28** Discuss the tradeoff between fairness and throughput of operations in the readers–writers problem. Propose a method for solving the readers–writers problem without causing starvation.
5.29 How does the signal() operation associated with monitors differ from the corresponding operation defined for semaphores?

5.30 Suppose the signal() statement can appear only as the last statement in a monitor function. Suggest how the implementation described in Section 5.8 can be simplified in this situation.

5.31 Consider a system consisting of processes $P_1, P_2, ..., P_n$, each of which has a unique priority number. Write a monitor that allocates three identical printers to these processes, using the priority numbers for deciding the order of allocation.

5.32 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.

5.33 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?

5.34 Suppose we replace the wait() and signal() operations of monitors with a single construct await(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 await statement so that it can be implemented efficiently? (Hint: Restrict the generality of B; see [Kessels (1977)].)

5.35 Design an algorithm for 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 invokes a function tick() in your monitor at regular intervals.

Programming Problems

5.36 Programming Exercise 3.20 required you to design a PID manager that allocated a unique process identifier to each process. Exercise 4.20 required you to modify your solution to Exercise 3.20 by writing a program that created a number of threads that requested and released process identifiers. Now modify your solution to Exercise 4.20 by ensuring that the data structure used to represent the availability of process identifiers is safe from race conditions. Use Pthreads mutex locks, described in Section 5.9.4.