Chapter 8
Process Synchronization

This chapter will discuss the following concepts:

8.1 Background

Concurrent access to shared data may result in data inconsistency. In this chapter, we discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

8.2 The Critical-Section Problem

Consider a system consisting of n processes {P0, P1, ..., Pn-1}. Each process has a segment of code, called a **critical section**, in which the process may be changing common variables, updating a table, writing a file, and so on.

The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. That is, **no two processes are executing in their critical sections at the same time**. The critical-section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section**. The general structure of a typical process P, is shown in Figure 8.1. The entry section and exit section are enclosed in boxes to highlight these important segments of code.

```
do{

    entry section

        critical section

    exit section

        remainder section

} while  (TRUE);
```

Figure 8.1 General structure of a typical process P.

A solution to the critical-section problem must satisfy the following three requirements:

1. Mutual exclusion. If process P; is executing in its critical section, then no other processes can be executing in their critical sections.

2. **Progress.** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.

3. Bounded waiting. There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

We assume that each process is executing at a nonzero speed.

8.3 Synchronization Hardware

In the following discussions, we explore several more solutions to the critical-section problem using techniques ranging from hardware to software based APIs available to application programmers. Hardware features can make any programming task easier and improve system efficiency. In this section, we present some simple hardware instructions that are available on many systems and show how they can be used effectively in solving the critical-section problem.

The critical-section problem could be solved simply in a uniprocessor environment if we could prevent interrupts from occurring while a shared variable was being modified. In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable. This is the approach taken by non-preemptive kernels.

Unfortunately, this solution is not as feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming, as the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases. Also, consider the effect on a system's clock, if the clock is kept updated by interrupts.

Many modern computer systems therefore provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words atomically—that is, as one uninterruptible unit. We can use these special instructions to solve the critical-section problem in a relatively simple manner. Rather than discussing one specific instruction for one specific machine, we abstract the main concepts behind these types of instructions.

1. The TestAndSet ( ) instruction
2. The Swap ( ) instruction,

8.4 Semaphores

The various hardware-based solutions to the critical-section problem (using the TestAndSet( ) and Swap( ) instructions) presented in Section 8.3 are complicated for application programmers to use. To overcome this difficulty, we can use a synchronization tool called a **semaphore**.

A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: **wait** ( ) and **signal** ( ).

The definition of wait ( ) is as follows:

```
wait(S) {
     while S <=0
          ; // no-op
     S--;
}
```

The definition of signal ( ) is as follows:

```
signal(S) {
     S++;
}
```

All the modifications to the integer value of the semaphore in the wait ( ) and signal ( ) operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of wait(S), the testing of the integer value of S (S <= 0), and its possible modification (S--), must also be executed without interruption. let us see how semaphores can be used.

8.4.1 Usage

Operating systems often distinguish between **counting** and **binary** semaphores. The value of a counting semaphore can range over an unrestricted domain. The value of a binary semaphore can range only between 0 and 1. On some systems, binary semaphores are known as **mutex locks**, as they are locks that provide mutual exclusion.

We can use binary semaphores to deal with the critical-section problem for multiple processes. The n processes share a semaphore, mutex initialized to 1. Each process P is organized as shown in Figure 8.2.

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a wait ( ) operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a signal ( ) operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

```
do {

   waiting(mutex);

      // critical section

   signal(mutex);
      // remainder section
}while (TRUE);
```

Figure 8.2 Mutual-exclusion implementation with semaphores.

8.5 Classic Problems of Synchronization

1 The Bounded-Buffer Problem

2 The Readers-Writers Problem

3 The Dining-Philosophers Problem

End of chapter 8