Lecture 7
Process Management: Process Synchronization

Overview

- Consider the producer–consumer problem, with a bounded buffer used to enable processes to share memory:

The code for producer:

```
while (true) {
     /* produce an item in next_produced */

     while (counter == BUFFER_SIZE)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
     counter++;
}
```

The code for consumer:

```
while (true) {
        while (counter == 0)
           ; /* do nothing */

        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;

        /* consume the item in next_consumed */
}
```

They may not function correctly when executed concurrently:

As an illustration, suppose that the value of the variable counter is currently 5 and that the producer and consumer processes concurrently execute the statements “counter++” and “counter--”. Following the execution of these two statements, the value of the variable counter may be 4, 5, or 6! The only correct result is counter == 5, which is generated correctly if the producer and consumer execute separately.

- We can show that the value of counter may be incorrect as follows. Note that the statement “counter++” and “counter--” may be implemented in machine language (on a typical machine) as follows:

register1 = counter

register1 = register1 + 1

counter = *register1*

register2 = counter

register2 = register2 − 1

counter = *register2*

The concurrent execution of “counter++” and “counter--”:

T0: producer execute register1 = counter {register1 = 5}

T1: producer execute register1 = register1 + 1 {register1 = 6}

T2: consumer execute register2 = counter {register2 = 5}

T3: consumer execute register2 = register2 − 1 {register2 = 4}

T4: producer execute counter = register1 {counter = 6}

T5: consumer execute counter = register2 {counter = 4}

- We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently.
- A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**.
- To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable counter. To make such a guarantee, we require that the processes be synchronized in some way.

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 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**.

```
do {
        entry section
                critical section
        exit section
                remainder section
} while (true);
```

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

1. Mutual exclusion. If process Pi 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 deciding 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.

Peterson's Solution

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

Figure 5.2 The structure of process P_i in Peterson's solution.

Peterson’s solution is restricted to two processes that alternate execution between their critical sections and remainder sections.

Peterson’s solution requires the two processes to share two data items:

int turn;

boolean flag[2];

- The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i, then process Pi is allowed to execute in its critical section. The flag array is used to indicate if a process is ready to enter its critical section.

We now prove that this solution is correct. We need to show that:

1. **Mutual exclusion is preserved.**
2. **The progress requirement is satisfied.**
3. **The bounded-waiting requirement is met.**

To prove property 1, we note that each Pi enters its critical section only if either flag[j] == false or turn == i.

Also note that, if both processes can be executing in their critical sections at the same time, then flag[0] == flag[1] == true.

These two observations imply that P0 and P1 could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both.

Hence, one of the processes —say, Pj—must have successfully executed the while statement, whereas Pi had to execute at least one additional statement (“turn == j”).

However, at that time, flag[j] == true and turn == j, and this condition will persist as long as Pj is in its critical section; as a result, mutual exclusion is preserved.

To prove properties 2 and 3, we note that a process Pi can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag[j] == true and turn == j; this loop is the only one possible. If Pj is not ready to enter the critical section, then flag[j] == false, and Pi can enter its critical section.

If Pj has set flag[j] to true and is also executing in its while statement, then either turn == i or turn == j. If turn == i, then Pi will enter the critical section. If turn == j, then Pj will enter the critical section.

However, once Pj exits its critical section, it will reset flag[j] to false, allowing Pi to enter its critical section. If Pj resets flag[j] to true, it must also set turn to i.

Thus, since Pi does not change the value of the variable turn while executing the while statement, Pi will enter the critical section (progress) after at most one entry by Pj (bounded waiting).

Mutex Locks

We use the **mutex lock** to protect critical regions and thus prevent race conditions. That is, a process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section. The acquire()function acquires the lock, and the release() function releases the lock.

```
acquire() {
while (!available)
        ; /* busy wait */
available = false;
        }
```

```
release() {
 available = true;
      }
```

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (true);
```

The main disadvantage of the implementation given here is that it requires **busy waiting.** While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire(). In fact, this type of mutex lock is also called a **spinlock.** This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively.

Semaphores

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

```
wait(S) {
        while (S <= 0)
         ; // busy wait
         S--;
        }

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

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. Thus, binary semaphores behave similarly to mutex locks.

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.

We can also use semaphores to solve various synchronization problems.

For example, consider two concurrently running processes: *P1 with a statement S1 and P2 with a statement S2. Suppose we require that S2 be executed only* after *S1 has completed. We can implement this scheme readily by letting P1* and *P2 share a common semaphore synch, initialized to 0. In process P1, we* insert the statements

S1;

signal(synch);

In process *P2, we insert the statements*

wait(synch);

S2;

Deadlocks and Starvation

P0	*P1*
wait(S);	wait(Q);
wait(Q);	wait(S);
. .	
. .	
. .	
signal(S);	signal(Q);
signal(Q);	signal(S);

Suppose that *P0 executes wait(S) and then P1 executes wait(Q). When P0* executes wait(Q), it must wait until *P1 executes signal(Q). Similarly, when P1 executes wait(S), it must wait until P0 executes signal(S). Since these* signal() operations cannot be executed, *P0 and P1 are* ***deadlocked****.*

We say that a set of processes is in a **deadlocked state** when every process in the set is waiting for an event that can be caused only by another process in the set.

Another problem related to deadlocks is **indefinite blocking or starvation,** a situation in which processes wait indefinitely within the semaphore. Indefinite blocking may occur if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

Classic Problems of Synchronization

There are a number of synchronization problems as examples of a large class of concurrency-control problems. In the solutions to the problems, semaphores for synchronization is used. These problems are:

The Bounded-Buffer Problem

The Readers–Writers Problem

The Dining-Philosophers Problem

The Dining-Philosophers Problem

The **dining-philosophers problem** is considered a classic synchronization problem because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore. She releases her chopsticks by executing the signal() operation on the appropriate semaphores.

Figure 5.13 The situation of the dining philosophers.

Thus, the shared data are semaphore chopstick[5]; where all the elements of chopstick are initialized to 1. The structure of philosopher i is:

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for awhile */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
} while (true);
```

Figure 5.14 The structure of philosopher *i*.

Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock. Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Several possible solutions to the deadlock problem are available:

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
- Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even numbered philosopher picks up her right chopstick and then her left chopstick.