Chapter 6: Process Synchronization

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true)

        /* produce an item and put in nextProduced
       while (count == BUFFER_SIZE)
               ; // do nothing
       buffer [in] = nextProduced;
       in = (in + 1) % BUFFER_SIZE;
       count++;
  }
```

Consumer

```
while (1)
 {
        while (count == 0)
              ; // do nothing
        nextConsumed =  buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        /*  consume the item in nextConsumed
 }
```

Race Condition

- count++ could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

- count-- could be implemented as

```
register2 = count
register2 = register2 - 1
count = register2
```

- Consider this execution interleaving with "count = 5" initially:

```
S0: producer execute register1 = count   {register1 = 5}
S1: producer execute register1 = register1 + 1   {register1 = 6}
S2: consumer execute register2 = count   {register2 = 5}
S3: consumer execute register2 = register2 - 1   {register2 = 4}
S4: producer execute count = register1   {count = 6 }
S5: consumer execute count = register2   {count = 4}
```

Solution to Critical-Section Problem

1. Mutual Exclusion - If process P_i 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 there exist some processes that wish to enter their critical section, only those processes not executing in their remainder sections can participate in the decision on which process will enter its critical section next.
3. Bounded Waiting - A bound must exist 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

The Critical Section Problem (1/2)

- Two process solution: P_i and P_j
- Pseudo-code of process P_i, the one of P_j is symmetric.

```
do {
          while (turn == j); %wait

          CRITICAL SECTION

          turn = j;

          REMAINDER SECTION

     } while (TRUE);
```

- Problem: This algorithm does NOT satisfy the progress requirement because …

The Critical Section Problem (2/2)

- Two process solution: P_i and P_j
- Pseudo-code of process P_i, the one of P_j is symmetric.

```
do {
        flag[i] = TRUE;
        while(flag[j]); %wait only if also Pj wants to enter the critical section

        CRITICAL SECTION

        flag[i] = FALSE;

        REMAINDER SECTION

    } while (TRUE);
```

- Problem: Deadlock

Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
    - int turn;
    - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready!

Algorithm for Process P_i

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

              CRITICAL SECTION

        flag[i] = FALSE;

                REMAINDER SECTION

    } while (TRUE);
```

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

Semaphore

- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
- Can only be accessed via two indivisible (atomic) operations

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

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

Semaphore as General Synchronization Tool

- Provides mutual exclusion
  - Semaphore S;   // initialized to 1
  - wait (S);

```
        Critical Section
signal (S);
```

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
  - S.value: integer
  - S.L: pointer to next record in the list
- Two operations:
  - block – place the process invoking the operation on the appropriate waiting queue.
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue.

Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of wait:

```
wait (S){
        S.value--;
        if (S.value < 0) {
                add process P to waiting queue S.L
                block(P);  }
}
```

- Implementation of signal:

```
Signal (S){
        S.value++;
        if (S.value <= 0) {
                remove process P from the waiting queue S.L
                wakeup(P);  }
}
```

Deadlock and Starvation

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

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

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

- *N* buffers, each can hold one item
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N.

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {

        //   produce an item

     wait (empty);
     wait (mutex);

          //  add the item to the  buffer

      signal (mutex);
      signal (full);
  } while (true);
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
do {
    wait (full);
    wait (mutex);

        //  remove an item from  buffer

     signal (mutex);
     signal (empty);

         //  consume the removed item

} while (true);
```

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Solution: monitors

Linux Synchronization

- Linux:
  - disables interrupts to implement short critical sections
- Linux provides:
  - semaphores
  - spin locks