CSC 4103 - Operating Systems
Spring 2007

LECTURE - VI

PROCESS SYNCHRONIZATION - II

Tevfik Koşar

Louisiana State University
February 8th, 2007

Bounded Buffer Problem

- 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);
```

Bounded Buffer – 1 Semaphore Soln

- The structure of the producer process

```
 int empty=N, full=0;
do {
     //   produce an item
    wait (mutex);
             if (empty> 0){
                 //  add the item to the  buffer
                 empty --; full++;
             }
    signal (mutex);

  } while (true);
```

Bounded Buffer – 1 Semaphore Soln

- The structure of the consumer process

```
do {

    wait (mutex);
        if (full>0){
          //  remove an item from  buffer
          full--; empty++;
        }
    signal (mutex);

    //  consume the removed item

} while (true);
```

5

Bounded Buffer – 1 Semaphore Soln - II

- The structure of the producer process

```
 int empty=N, full=0;
do {
   //   produce an item
  while (empty == 0){}
  wait (mutex);
        //  add the item to the  buffer
        empty --; full++;
  signal (mutex);

 } while (true);
```

6

Bounded Buffer – 1 Semaphore Soln - II

- The structure of the consumer process

```
do {
      while (full == 0){}
       wait (mutex);
            //  remove an item from  buffer
            full--; empty++;
       signal (mutex);

       //  consume the removed item


} while (true);
```

7

Bounded Buffer – 2 Semaphore Soln

- The structure of the producer process

```
do {

          //   produce an item
    wait (empty);
        //  add the item to the  buffer
    signal (full);

  } while (true);
```

8

Bounded Buffer – 2 Semaphore Soln

- The structure of the consumer process

```
do {
    wait (full);
        //  remove an item from  buffer
    signal (empty);

        //  consume the removed item

} while (true);
```

9

Readers-Writers Problem

- The structure of a writer process

```
do  {
      wait (wrt) ;

        //    writing is performed

      signal (wrt) ;
  } while (true)
```

10

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
     wait (mutex) ;
     readercount ++ ;
     if (readercount == 1)  wait (wrt) ;
     signal (mutex)

          // reading is performed

     wait (mutex) ;
     readercount  - - ;
     if readercount  == 0)  signal (wrt) ;
     signal (mutex) ;
 } while (true)
```

11

Dining Philosophers Problem

• Five philosophers spend their time eating and thinking.

• They are sitting in front of a round table with spagett served.

•There are five plates at the table and five chopsticks set between the plates.

• Eating the spaghetti requires the use of two chopsticks which the philosophers pick up one at a time.

•Philosophers do not talk to each other.

•Semaphore chopstick [5] initialized to 1

12

Dining-Philosophers Problem (Cont.)

- The structure of Philosopher ***i***:

```
Do {
    wait ( chopstick[i] );
    wait ( chopStick[ (i + 1) % 5] );

        //  eat

    signal ( chopstick[i] );
    signal (chopstick[ (i + 1) % 5] );

        //  think

} while (true) ;
```

To Prevent Deadlock

- Ensures mutual exclusion, but does not prevent deadlock
- Allow philosopher to pick up her chopsticks only if both chopsticks are available (i.e. in critical section)
- Use an asymmetric solution: an odd philosopher picks up first her left chopstick and then her right chopstick; and vice versa

Disjkstra’s Solution

For each philosopher i:

1. Think
2. Wait(room_semaphore) //max value = 4
3. Wait(left_fork_i)
4. Wait(right_fork_i)
5. Eat spaghetti
6. Signal(left_fork_i)
7. Signal(right_fork_i)
8. Signal(room_semaphore)
9. Repeat 1.

* Only four philosophers would be able to enter the room simultaneously and no deadlock can occur.

Chandy/Misra Solution (1984)

- For every pair of philosophers contending for a resource, create a fork and give it to the philosopher with the lower ID. Each fork can either be *dirty* or *clean*. Initially, all forks are dirty.
- When a philosopher wants to use a set of resources (*i.e.*, eat), it must obtain the forks from its contending neighbors. For all such forks it does not have, it sends a request message.
- When a philosopher with a fork receives a request message, it keeps the fork if it is clean, but gives it up when it is dirty. If it sends the fork over, it cleans the fork before doing so.
- After a philosopher is done eating, all its forks become dirty. If another philosopher had previously requested one of the forks, it cleans the fork and sends it.

Sleeping Barber Problem

- Based upon a hypothetical barber shop with one barber, one barber chair, and a number of chairs for waiting customers
- When there are no customers, the barber sits in his chair and sleeps
- As soon as a customer arrives, he either awakens the barber or, if the barber is cutting someone else's hair, sits down in one of the vacant chairs
- If all of the chairs are occupied, the newly arrived customer simply leaves

Solution

- Use three semaphores: one for any waiting customers, one for the barber (to see if he is idle), and a mutex
- When a customer arrives, he attempts to acquire the mutex, and waits until he has succeeded.
- The customer then checks to see if there is an empty chair for him (either one in the waiting room or the barber chair), and if none of these are empty, leaves.
- Otherwise the customer takes a seat – thus reducing the number available (a critical section).
- The customer then signals the barber to awaken through his semaphore, and the mutex is released to allow other customers (or the barber) the ability to acquire it.
- If the barber is not free, the customer then waits. The barber sits in a perpetual waiting loop, being awakened by any waiting customers. Once he is awoken, he signals the waiting customers through their semaphore, allowing them to get their hair cut one at a time.

Implementation:

+ Semaphore Customers
+ Semaphore Barber
+ Semaphore accessSeats (mutex)
+ int NumberOfFreeSeats

The Barber(Thread):

```
while(true) //runs in an infinite loop
{
  Customers.wait() //tries to acquire a customer - if none is available he's going to
     sleep
  accessSeats.wait() //at this time he has been awaken -> want to modify the number
     of available seats
  NumberOfFreeSeats++ //one chair gets free
  Barber.signal() // the barber is ready to cut
  accessSeats.signal() //we don't need the lock on the chairs anymore //here the
     barber is cutting hair
}
```

19

The Customer(Thread):

```
while (notCut) //as long as the customer is not cut
{
  accessSeats.wait() //tries to get access to the chairs
  if (NumberOfFreeSeats>0) { //if there are any free seats
    NumberOfFreeSeats -- //sitting down on a chair
    Customers.signal() //notify the barber, who's waiting that there is
    a customer
    accessSeats.signal() // don't need to lock the chairs anymore
    Barber.wait() // now it's this customers turn, but wait if the barber
    is busy
    notCut = false
  } else // there are no free seats //tough luck
  accessSeats.signal() //but don't forget to release the lock on the
    seats }
```

20

Any Questions?

21

Reading Assignment

- Read chapter 6 from Silberschatz.

22

Acknowledgements

- “Operating Systems Concepts” book and supplementary material by Silberschatz, Galvin and Gagne.

23