CSC 4103 - Operating Systems
Fall 2009

LECTURE - XI

DEADLOCKS - II

Tevfik Koşar

Louisiana State University

September 29th, 2009

Roadmap

- Classic Problems of Synchronization
  - Bounded Buffer
  - Readers-Writers
  - Dining Philosophers
  - Sleeping Barber
- Deadlock Prevention

Classical Problems of Synchronization

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

Bounded-Buffer Problem

- Shared buffer with N slots to store at most N items
- Producer processes data items and puts into the buffer
- Consumer gets the data items from the buffer
- Variable empty keeps number of empty slots in the butter
- Variable full keeps number of full items in the buffer

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);
```

consume non-existing item!

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);
```


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);
```

* Mutual Exclusion not preserved!

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);
```


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);
```

* Mutual Exclusion not preserved!

Bounded Buffer - 3 Semaphore Soln

- Semaphore mutex for access to the buffer, initialized to 1
- Semaphore full (number of full buffers) initialized to 0
- Semaphore empty (number of empty buffers) initialized to N

Bounded Buffer - 3 Semaphore Soln

- The structure of the producer process

```
do {

        //   produce an item

    wait (empty);
    wait (mutex);

       //  add the item to the  buffer

     signal (mutex);
     signal (full);
```

Bounded Buffer - 3 Semaphore Soln

- The structure of the consumer process

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

      //  remove an item from  buffer

    signal (mutex);
    signal (empty);

      //  consume the removed item
```


Readers-Writers Problem

- Multiple Readers and writers concurrently accessing the same database.
- Multiple Readers accessing at the same time --> OK
- When there is a Writer accessing, there should be no other processes accessing at the same time.

Readers-Writers Problem

- The structure of a writer process

```
do {
      wait (wrt) ;

        //    writing is performed

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


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)
```

Dining Philosophers Problem

• Five philosophers spend their time eating and thinking.

• They are sitting in front of a round table with spaghetti 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

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

Problems with Semaphores

- Wrong use of semaphore operations:
  - semaphores *A* and *B*, initialized to 1

P_0	P_1
wait (A);	*wait(B)*
wait (B);	*wait(A)*

➔Deadlock

  - signal (mutex) …. wait (mutex)
    ➔ violation of mutual exclusion
  - wait (mutex) … wait (mutex)
    ➔Deadlock
  - Omitting of wait (mutex) or signal (mutex) (or both)
    ➔ violation of mutual exclusion or deadlock

Semaphores

- inadequate in dealing with deadlocks
- do not protect the programmer from the easy mistakes of taking a semaphore that is already held by the same process, and forgetting to release a semaphore that has been taken
- mostly used in low level code, eg. operating systems
- the trend in programming language development, though, is towards more structured forms of synchronization, such as monitors and channels

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (…) { …. }
                …
    procedure Pn (…) {……}

     Initialization code ( ….) { … }
                …
    }
}
```

- A monitor procedure takes the lock before doing anything else, and holds it until it either finishes or waits for a condition

Monitor - Example

As a simple example, consider a monitor for performing transactions on a bank account.

```
monitor account {
    int balance := 0

  function withdraw(int amount) {
    if amount < 0 then error "Amount may not be negative"
    else if balance < amount then error "Insufficient funds"
    else balance := balance - amount
  }

  function deposit(int amount) {
    if amount < 0 then error "Amount may not be negative"
    else balance := balance + amount
  }
}
```


Condition Variables

- Provide additional synchronization mechanism
- condition x, y;

- Two operations on a condition variable:
  - x.wait () – a process invoking this operation is suspended
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()

If no process suspended, x.signal() operation has no effect.

Solution to Dining Philosophers using Monitors

```
monitor DP
  {
    enum { THINKING; HUNGRY, EATING) state [5] ;
    condition self [5];        //to delay philosopher when he is
                                        hungry but unable to get chopsticks

initialization_code() {
          for (int i = 0; i < 5; i++)
                    state[i] = THINKING;
    }


void pickup (int i) {
          state[i] = HUNGRY;
          test(i);//only if both neighbors are not eating
          if (state[i] != EATING) self [i].wait;
    }
```


Solution to Dining Philosophers (cont)

```
    void test (int i) {
          if ((state[i] == HUNGRY) &&
             (state[(i + 1) % 5] != EATING) &&
             (state[(i + 4) % 5] != EATING) ) {
                state[i] = EATING ;
               self[i].signal () ;
          }
     }

 void putdown (int i) {
          state[i] = THINKING;
                // test left and right neighbors
           test((i + 4) % 5);
           test((i + 1) % 5);
    }
}
```

- No two philosophers eat at the same time
- No deadlock
- But starvation can occur!

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
}
```


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 }
```

Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state.
  - ➔deadlock prevention or avoidance
- Allow the system to enter a deadlock state and then recover.
  - ➔deadlock detection
- Ignore the problem and pretend that deadlocks never occur in the system
  - ➔ Programmers should handle deadlocks (UNIX, Windows)

Deadlock Prevention

➔ Ensure one of the deadlock conditions cannot hold

➔Restrain the ways request can be made.

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources.
  - Eg. read-only files
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources.
  1. Require process to request and be allocated all its resources before it begins execution
  2. or allow process to request resources only when the process has none.

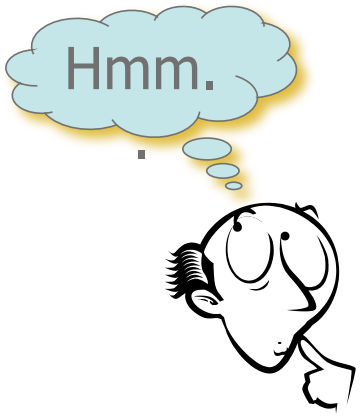
  Example: Read from DVD to memory, then print.
  1. holds printer unnecessarily for the entire execution
     - Low resource utilization
  2. may never get the printer later
     - starvation possible

Deadlock Prevention (Cont.)

- **No Preemption** –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
  - Preempted resources are added to the list of resources for which the process is waiting.
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

Summary

- Classic Problems of Synchronization
  - Bounded Buffer
  - Readers-Writers
  - Dining Philosophers
  - Sleeping Barber
- Deadlock Prevention



- Reading Assignment: Chapter 7 from Silberschatz.

Acknowledgements

- “Operating Systems Concepts” book and supplementary material by A. Silberschatz, P. Galvin and G. Gagne
- “Operating Systems: Internals and Design Principles” book and supplementary material by W. Stallings
- “Modern Operating Systems” book and supplementary material by A. Tanenbaum
- R. Doursat and M. Yuksel from UNR