CSC 4103 - Operating Systems
Spring 2008

LECTURE - VIII

PROCESS SYNCHRONIZATION

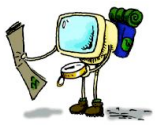
Tevfik Koşar

Louisiana State University

February 8th, 2007

Roadmap

- Process Synchronization
- The Critical-Section Problem
- Semaphores
- Classic Problems of Synchronization



Critical Section

- Critical section: segment of code in which the process may be changing shared data (eg. common variables)
- No two processes should be executing in their critical sections at the same time
- Critical section problem: design a protocol that the processes use to cooperate

Solution to Critical-Section Problem

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

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, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

Solution to Critical-Section Problem

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
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the N processes

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

- Semaphore *S* – integer variable
- Two standard operations modify wait() and signal()
  - Originally called P() and V()

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

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

- Less complicated
- Can only be accessed via two indivisible (atomic) operations

Semaphores as Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks

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

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

P_0	P_1
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 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 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);
```

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
```


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);
```

* Possibility of deadlock!

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!

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)
```


Summary

- Process Synchronization
- The Critical-Section Problem
- Semaphores
- Classic Problems of Synchronization

- Next Lecture: Deadlocks - I
- Reading Assignment: Chapter 6 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