CSC 4103 - Operating Systems
Spring 2007

LECTURE - VI

PROCESS SYNCHRONIZATION

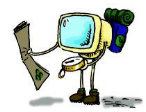
Tevfik Koşar

Louisiana State University

February 6th, 2007

Roadmap

- Process Synchronization
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Atomic Transactions



Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Consider consumer-producer problem:
  - 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}

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

7

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

8

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

9

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!

10

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);
```

11

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

12

TestAndndSet Instruction

- Definition:

```
boolean TestAndSet (boolean *target)
{
     boolean rv = *target;
     *target = TRUE;
     return rv:
}
```


Solution using TestAndSet

- Shared boolean variable lock initialized to false.
- Solution:

```
do {
   while ( TestAndSet (&lock ))
           ;   /* do nothing

      //    critical section

   lock = FALSE;

      //      remainder section

} while ( TRUE);
```


Swap Instruction

- Definition:

```
void Swap (boolean *a, boolean *b)
{
     boolean temp = *a;
     *a = *b;
     *b = temp:
}
```


Solution using Swap

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key.
- Solution:

```
do {
     key = TRUE;
     while ( key == TRUE)
          Swap (&lock, &key );
        //    critical section
     lock = FALSE;
        //      remainder section
   } while ( TRUE);
```


Atomic TestAndSet and Swap

- Implementing atomic TestAndSet() and Swap() instructions on multiprocessors is not trivial at HW level
- Also complicated for application programmers for use

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);
```

19

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.

20

Classical Problems of Synchronization

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

21

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

22

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);
```

23

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);
```

24

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do **not** perform any updates
  - Writers – can both read and write.
- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1. (for readcount)
  - Semaphore wrt initialized to 1. (for writers)
  - Integer readcount initialized to 0.

Readers-Writers Problem (Cont.)

- 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) ;
     readcount ++ ;
     if (readercount == 1)  wait (wrt) ;
     signal (mutex)

          // reading is performed

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


Dining Philosophers Problem

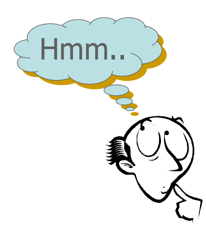
- Five philosophers spend their time eating and thinking.
- They are sitting in front of a round table with spaghetti served.
- There are five plates at the table and five forks set between the plates.
- Eating the spaghetti requires the use of two forks which the philosophers pick up one at a time.
- 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) ;
```


Any Questions?



Reading Assignment

- Read chapter 6 from Silberschatz.

Acknowledgements

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