CS323 Operating Systems
Process Synchronization

Yuanyuan Zhou
Lecture 9
2/10/2003

Content of this lecture

- Administrative announcements
- TSL
- Sleep and Wakeup
- Semaphores
- Monitor
- Barrier
- Summary

Administrative

- MP1 is due Friday, 11:59pm CST.

Review: Critical Regions

- What are Data Races
- Critical region and mutual exclusion
- Mutual exclusion using busy waiting
  - Disabling Interrupts
  - Lock Variables
  - Strict Alternation
  - Peterson's solution

Test & Set (TSL)

- Requires hardware support
- Does test and set atomically

```
char Test_and_Set ( char* target);
\\ All done atomicall
{    char temp = *target;
     *target = true;
     return(temp)
}
```


TSL instruction

```
enter_region:
    TSL REGISTER,LOCK        | copy lock to register and set lock to 1
    CMP REGISTER,#0          | was lock zero?
    JNE enter_region         | if it was non zero, lock was set, so loop
    RET | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0             | store a 0 in lock
    RET | return to caller
```

Other Similar Hardware Instruction

- Swap = TSL

```
void Swap (char* x,* y);
\\ All done atomically

{    char temp = *x;
     *x = *y;
     *y = temp
}
```


Sleep and Wakeup

- Problem with previous solutions
  - Busy waiting
  - Wasting CPU
  - Priority Inversion:
    - a high priority waits for a low priority to leave the critical section
    - the low priority can never execute since the high priority is not blocked.
- Solution: sleep and wakeup
  - When blocked, go to sleep
  - Wakeup when it is OK to retry entering the critical section

Producer-Consumer Problem (Problem?)

```
#define N 100                                /* number of slots in the buffer */
int count = 0;                               /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                          /* repeat forever */
          item = produce_item( );            /* generate next item */
          if (count == N) sleep( );          /* if buffer is full, go to sleep */
          insert_item(item);                 /* put item in buffer */
          count = count + 1;                 /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);  /* was buffer empty? */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                          /* repeat forever */
          if (count == 0) sleep( );          /* if buffer is empty, got to sleep */
          item = remove_item( );             /* take item out of buffer */
          count = count − 1;                 /* decrement count of items in buffer */
          if (count == N − 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                /* print item */
     }
}
```


Semaphores

- A semaphore count represents count number of abstract resources.
- New variable having 2 operations
  - The Down (P) operation is used to acquire a resource and decrements count.
  - The Up (V) operation is used to release a resource and increments count.
- Any semaphore operation is indivisible (atomic: what else we have talked before is atomic?)
- Semaphores solve the problem of the wakeup-bit

What's Up? What's Down?

- Definitions of P and V:

```
Down(S) {
    while (S <= 0)  { }; // no-op
    S= S-1;
    }

Up(S) {
   S++;
}
```

- Counting semaphores: 0..N
- Binary semaphores: 0,1

Mutex: Binary Semiphore

- Variable with only 2 states
  - Lock
  - Unlock
- Simplified version of semaphore
- Mutex is used for mutual exclusion

Mutex Implementation using TSL

- 2-3 person group discussion (2 minutes)
- Using Test_and_Set (TSL) instruction to implement
  - Mutex_lock
  - Mutex_unlock

Mutex Implementation using TSL

```
mutex_lock:
      TSL REGISTER,MUTEX              | copy mutex to register and set mutex to 1
      CMP REGISTER,#0                 | was mutex zero?
      JZE ok                          | if it was zero, mutex was unlocked, so return
      CALL thread_yield               | mutex is busy; schedule another thread
      JMP mutex_lock                  | try again later
ok:   RET | return to caller; critical region entered


mutex_unlock:
      MOVE MUTEX,#0                   | store a 0 in mutex
      RET | return to caller
```

Implementation of *mutex_lock* and *mutex_unlock*

Producer-Consumer Problem using Semaphores

```
semaphore mutex = 1;                /* controls access to critical region */
semaphore empty = N;                /* counts empty buffer slots */
semaphore full = 0;                 /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {                  /* TRUE is the constant 1 */
        item = produce_item( );     /* generate something to put in buffer */
        [redacted]                  /* decrement empty count */
                                    /* enter critical region */
        insert_item(item);          /* put new item in buffer */
        [redacted]                  /* leave critical region */
                                    /* increment count of full slots */
    }
}


void consumer(void)
{
    int item;

    while (TRUE) {                  /* infinite loop */
        [redacted]                  /* decrement full count */
                                    /* enter critical region */
        item = remove_item( );      /* take item from buffer */
        [redacted]                  /* leave critical region */
                                    /* increment count of empty slots */
        consume_item(item);         /* do something with the item */
    }
}
```


Using Mutex to Implement Semaphores

- 2-3 person group discussion (2 minutes)
- Using mutex_lock and mutex_unlock to implement a counter semaphore
  - Up
  - Down

Busy Wait Semaphore Implementation (1)

```
class Semaphore {
      Mutex m;    // Mutual exclusion.
      int count;    // Resource count.
public:
      Semaphore( int count );
      void Up();
      void Down();
};

static inline Semaphore::Semaphore( int count )
{
      count = count;
}
```


Busy Wait Semaphore Implementation(2)

```
void
Semaphore::Down()
{   mutex_lock(m);
    while ( count == 0 )
    {
      mutex_unlock(m);
      yeild();
      mutex_lock(m);
    }
    count--;
    mutex_unlock();
}
```

```
void
Semaphore::Up()
{
    mutex_lock(m);
    count++;
    mutex_unlock(m);
}
```

Implementations of Semaphore using Sleep & Wakeup

```
type Semaphore = record
        value:integer;
        L: list of processes;
Semaphore S;

Down(S):
  S.value:= S.value - 1;
  if S. value < 0 then
   {
     add this process to the S.L;
     block;
   };

Up(S):
  S.value:= S.value + 1;
  if S.value > 0 then
   {
     remove process P from S.L;
     wakeup(P);
   }
```

Does it work?

Tradeoffs

- Busy waiting (spinlock)
  - Waste CPU cycles
- Sleep&Wakeup (blocked lock)
  - Context switch overhead
- Hybrid competitive solution (spin-block)
  - Apply spinlocks if the waiting time is shorter than the context switch time
  - Use sleep & wakeup if the waiting time is longer than the context switch time

Possible Deadlocks with Semaphores

Example:

```
P0                          P1
  share two semaphores S and Q
  S:= 1; Q:=1;

wait(S); // S=0 ------------> wait(Q); //Q=0
wait(Q); // Q= -1  <--------
         -------------------->  wait(S); // S=-1
// P0 blocked                // P1 blocked

          DEADLOCK
signal(S);              signal(Q);
signal(Q);              signal(S);
```


Be Careful When Using Semaphores

```
// Violation of Mutual Exclusion
  signal(mutex);          mutexUnlock();
  critical section        criticalSection();
  wait(mutex);            mutexLock();

  // Deadlock Situation
  wait(mutex);            mutexLock();
  critical section        criticalSection();
  wait(mutex);            mutexLock(P);

  // Violation of Mutual Exclusion (omit wait(mutex)/mutexLock())

  critical section        critical Section();
  signal(mutex);          mutexUnlock();

  // Deadlock Situation (omit signal(mutex)/mutexUnlock())

  wait(mutex);            mutexLock();
  critical section        criticalSection();
```


Monitor

- A simpler way to synchronize
- A set of programmer defined operators

```
monitor  monitor-name
{

//   variable declaration

public  entry P1(..);
 {... };

......

public  entry Pn(..);
  {...};

begin
  initialization code
end
```

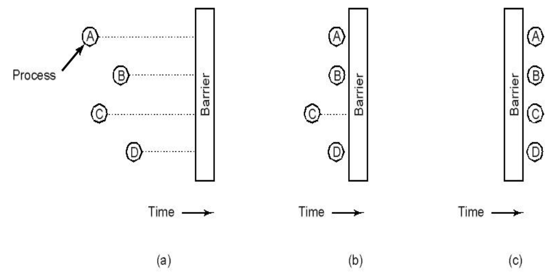

Monitor Properties

- The internal implementation of a monitor type cannot be accessed directly by the various threads.
- The encapsulation provided by the monitor type limits access to the local variables only by the local procedures.
- Monitor construct does not allow concurrent access to all procedures defined within the monitor.
- Only one thread/process can be active within the monitor at a time.
- Synchronization is built in.

Barriers

- Use of a barrier
  - processes approaching a barrier
  - all processes but one blocked at barrier
  - last process arrives, all are let through
- Problem:
  - Waste CPU if workloads are unbalanced

Barriers



Next lecture: Classic Problems

- Producer-Consumer problem
- Bounded buffer problem
- First Reader-writer problem
- Dining philosophers problem

Reminder

- MP1 due this Friday