Topics in Processes Synchronizaion

Contents

1. Partial Defintion of class Semaphore

```
class Semaphore {
   private int counter;
   public Semaphore(int v) {
         counter = v;
    }// end Semaphore()
   public void up() {
                 // increment counter; if counter
                 // negative or zero then signal
                 //waiting processes
    }//end up()
   public void down() {
                 // decrement counter.if counter is
            //negative then block and join the
            // waiting  (blocked) processes
                 // When wakeup, continue
```

```
        }//end down()
}// end class semaphor
```

2. Producer Consumer

2a. **Class Buffer**

```
class Buffer {
  private Object[] elements;
  private int size, nextIn, nextOut;
  Buffer(int size)   {
      this.size = size;
      elements = new Object[size];
       nextIn = 0;        nextOut = 0;
   }// end Buffer()
   public void addElement(Object o) {
       elements[nextIn++] = o;
       if (nextIn == size)     nextIn = 0;
    }//end addElement()

    public Object removeElement() {
    Object result =
                    elements[nextOut++];
     if (nextOut == size)   nextOut = 0;
     return result;
     }// end removeElement()
}// end class Buffer
```

2b. **Solution to Producer Consumer Problem using Semaphores**

```
shared Buffer b = new Buffer(10);
Semaphore mutex = 1,
              empty = 10,  full = 0;
```

```
class Producer implements Runnable {
    Object produce() { /* ... */ }

public void run() {
          Object item;
           for (;;) {
               item = produce();
               empty.down();
               mutex.down();
               b.addElement(item);
               mutex.up();
               full.up();
          }// end for()
     }// end run()
}// end class Producer

class Consumer implements Runnable{
      void consume(Object o) { /* ... */ }
            public void run() {
            Object item;
            for (;;) {
                full.down();
                mutex.down();
                item = b.removeElement();
                mutex.up();
                empty.up();
                consume(item);
            }//end for()
        }// end run()
}//end class Consumer
```

2c. Solution to Producer Consumer Problem Using Monitor

```
monitor BoundedBuffer {
  private Buffer b = new Buffer(10);
  private int count = 0;
  private Condition notfull, notempty;
  public void insert(Object item) {
      if (count == 10)  notfull.wait();
      b.addElement(item);
       count++;
        notempty.signal();
   }// end insert()

public Object remove() {
       if (count == 0)     notempty.wait();
       item result = b.removeElement();
       count--;
        notfull.signal();
        return result;
   }// end remove()
}//end Monitor
```

2d. Solution to Producer Consumer Problem in Java

```
class BoundedBuffer {
   private Buffer b = new Buffer(10);
   private int count = 0;
   synchronized public void
                        insert(Object  item) {
   while (count == 10)         wait();
    b.addElement(item);
```

```
     count++;
     notifyAll();
     }//end insert

synchronized public Object remove() {
     while (count == 0)            wait();
      Object result = b.removeElement();
      count--;
       notifyAll();
       return result;
        }// end remove()
    }//end class BoundedBuffer
```

3. Implementation

3a. Implementing Monitors by using Semaphores.

Operations on semaphore *sem* *sem*.down() and *sem*.up(), are assumed as given.

Part I: Monitor that uses the strategy of signalling threads waiting on condition variable and blocking the signaller.

1. The Compiler puts a "starter" to each method in the Monitor, using a semaphor *mutex* (initially 1):

mutex.down()

This semaphore will be raised on exit
Of the function (see below)

2. **Operations on condition variable x:**

```
// blocking on x is via xSem semaphore.
// xCount counts number of  threads
// blocked on x.
//signallers block themselves (within the // monitor) via highPriority semaphore.
// highCount counts number of threads
// blocked within the monitor.
//All counters and semaphores initially 0
```

x.wait() is implemented as:

```
  xCount++; // increase num of blocked
  if (highCount > 0) highPriority.up();
  else   mutex.up(); open the monitor
  xSem.down(); //block yourself
  xCount--; // on wakeup, decrease count
```

x.signal() is implemented as:

```
        // if no thread is waiting - no action
   if (xCount > 0) {
   highCount++;
   xSem.up(); // wake someone
   highPriority.down(); // wait here!
  highCount--;on wakeup, decrease count
}// end if
```

3. The compiler attaches to each method in the Monitor an exit code at the end:

```
  if (highCount > 0)  highPriority.up();
  else mutex.up();
```

PART II: Monitor that uses the strategy of <u>notifying</u> threads waiting on condition variable, and blocking the <u>waiters.</u>

1. The Compiler puts a "starter" to each method, using a semaphor *mutex* (initially 1):

mutex.down()

2. **Operations on condition variable x:**

```
// blocking on x is via xSem semaphore.
// xCount counts number of  threads
// blocked on x.
// The counter and semaphore initially 0
```

x.wait() is implemented as:

```
        xCount++;
    mutex.up();
                //alow others to enter  monitor
        xSem.down(); //block yourself
        mutex.down();
                //when notified, try to enter the
                //monitor, you might be blocked,
                // because the signaller continues
                // within the monitor!
```

```
x.notify() is implemented as:
// if no one waits - no action!
        if (xCount > 0) {
       xCount--;
       xSem.up();
          // wakeup others. Do not block
          // yourself.
   }//end if
```

3. The compiler attaches to each method in the Monitor an exit code at the end:

```
        mutex.up();
```

3b. Implementing Semaphores using Critical Section Primitives

Assume we have:

1. primitives begin_cs, end_cs that guarantee mutual exclusion to (uninterruptible, short) code sections between them.

2. function swap_process(PCB pcb).

This function takes a pointer to a PCB. It puts the state of the running process into pcb and the CPU starts running the process whose state was the value previously pointed to by pcb.

We have deffintions of PCB (process control block) and PCB_queue

```
class Semaphore {
  private PCB_queue waiters;
   // processes waiting for this Semaphore
   private int value;
         // if negative, number of waiters
  private static PCB_queue ready_list;
          // list of all processes ready to run

   Semaphore(int initialValue) {
        value = initialValue;
   }

public void down() {
      begin_cs
      value--;
       if (value < 0) {
            // The current process must wait
            // Find some other process to run.
            //The ready_list must be non
           //empty or there is a global
           //deadlock.
          PCB pcb =
                   ready_list.removeElement();
          swap_process(pcb);
          // Now pcb contains the state of
         //the process that called down(),
         //and the currently running process
         //is some other process.
         waiters.addElement(pcb);
```

```
    }// end if
    **end_cs**
  }// end down()

   public void up() {
     **begin_cs**
     value++;
      if (value <= 0) {
         // The value was previously
         //negative, so there is some
         //process waiting.  We must wake
         //it up.

         PCB pcb =
                 waiters.removeElement();
      ready_list.addElement(pcb);
    }//end if
    **end_cs**
  }// end up()
} // end class Semaphore
```

The implementation of swap_process()

This is usually is in assembly (we have to access registers)

We will describe it in Java, assume the CPU's current stack-pointer register is accessible as "SP".

```
void swap_process(PCB pcb) {
    int new_sp = pcb.saved_sp;
    pcb.saved_sp = SP;
    SP = new_sp;
}// end swap_process();
```

3c. Implementing Critical Sections Primitives

Note: We assume that if we have one CPU then interrupts are disabled throughout the (short) Critical Sections, so that no process switching occur.

If we have multiple CPUs, with shared memory, the Bus (or memory) hardware Serializes read/write word requests, so that they are executed atomically, and no further hardware support is required.

The **begin_cs()** and **end_cs()** primitives provides mutual exclusion to the Critical Sections. These primitives themselves **are not** uninterruptible.

Description: (Lamport Bakery Algorithm)
begin_cs(i):

Process i wants to get into the critical section (wants "to get a service").
It looks on the currently assigned ticket numbers, and writes on its ticket the highest number plus 1.

Due to concurrency, it may be that two processes will get equal numbers.

The process with the lowest ticket number (or in case of tie, the one with lowest process id), goes on to the Critical
Section ("gets a service")

All tickets are shared for reading.

end_cs(i)
Process i writes on his ticket the value 0.

Actual Implementation:

Shared memory:

```
static final int N = ...; // num of processes
shared boolean choosing[] = {
                    false, false, ..., false };

// choosing[j] == TRUE means that
  //process j is concurrently choosing
  // a number

  shared int ticket[] = { 0, 0, ..., 0 };



void begin_cs(int i) {
   choosing[i] = true;

   ticket[i] = 1 + max(ticket[0], ...,
                               ticket[N-1]);
   choosing[i] = false; //finished choosing
   for (int j=0; j<N; j++) {
          while (choosing[j]) ;
                   // wait until j finished choosing
          while (ticket[j] != 0
                   &&
          ( (ticket[j] < ticket[i]) ||
        ((ticket[j] == ticket[i]) && (j < i)) )
           ) ;
         // wait until process j finishes
         // (if he has lower number)
                   // then ticket[j] == 0
```

```
    }//end for - end waiting. Go on to C.S.
}// end begin_cs()

void end_cs(int i) {
      ticket[i] = 0;// finished.
 }//end end_cs()
```