Lecture 5: Synchronization and Mutual Exclusion

Summary

1. Computer system overview (Chapter 1).
2. Basic of virtual memory; i.e. segmentation and paging (Chapter 7 and part of 8).
3. Process (Chapter 3).
4. Mutual Exclusion and Synchronization (Chapter 5 section 1 and 3).
   - Conditions for race avoidance.
   - Strict alternation.
   - Semaphores (see Figure 3).
   - Producers and Consumers problem.

Concurrency (continued)

1. Key terms:
   - *Deadlock*. A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.
   - *Livelock*. A situation in which two or more processes continuously change their state in response to changes in the other processes without doing any useful work.
   - *Starvation*. A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.
2. Hardware support for mutual exclusion.
   - Disabling interrupt (see Figure 1).
   - Test-Set-Lock (TSL) (e.g. TSL RX, LOCK).
     - TSL copies the content of $LOCK$ to RX and then set $LOCK$ to a non-zero value (see Figure 4).
   - Exchange Instruction (see Figure 2).

Monitor (Chapter 5.4)

1. Monitor: A software module consisting of one or more procedures, and initialization sequence, and local data (see Figure 5).
2. Main characteristics of a monitor are the following:
   (a) The local data variables are accessible only by the monitor's procedures and not by any external procedure.
   (b) A process enters the monitor by invoking one of its procedures.
   (c) Only one process may be executing in the monitor at a time; any other process that has invoked the monitor is blocked, waiting for the monitor to become available.

- Two or more calls to two different procedures in the monitor? NO!
- Two or more calls to the same proceduresin the monitor? NO!

3. Condition variables: Special data type contained in the monitor and accessible only within the monitor. The variable is operated on by two functions:

   - *cwait(cv)*: Suspend execution of the calling process or thread on condition variable cv. The monitor is now available for use by another process.
   - *csignal(cv)*: Resume execution of some process blocked after a $cwait$ on the same condition variable (cv). If there are several such processes or threads, choose one of them; if there are no waiting processes, do nothing (note that this is different than semaphore).

4. Signaling disciplines (see Figure 6).

   - Signal and continue: signalers continues and the signaled process or thread go to entry queue (non-premptive).
   - Signal and wait: signalers goes to the entry queue and the signaled process or thread executes now (preemptive).

5. Solving producer/consumer problem with monitors (see Figure 7).

```
while(1)
{
        __asm__ __volatile__ ("cli");  /* disabling interrupt */
        /* critical section */
        __asm__ __volatile__ ("sti");  /* enabling interrupt */
        /* remainder */
}
```

Figure 1: Enabling and Disabling Interrupt in X-86 Processors.

```
/*Mutual exclusion with XCHG instruction */
#define N 4

int const n = N /* number of processes or threads */
int bolt;

void P(int i)
{
        int keyi = 1;
        while (1)
        {
                do exchange(keyi, bolt)
                while (keyi != 0)
                /* critical section */;
                exchange (keyi, bolt);
                /* non-critical section */
        }
}

void main()
{
        bolt = 0;
        parbegin(P(1), P(2), ..., P(n));
}

void exchange (int key, int lock)
{
        int temp;
        temp = lock;
        lock = key;
        key = temp;
}
```

Figure 2: Exchange instruction in X-86 Processors.

```
/* From Andrew Tanenbaum's Modern Operating Systems, 2nd Edition */
#define N 10000                 /* Buffer size */
typedef int semaphore;
semaphore mutex = 1;            /* Access to critical section */
semaphore empty = N;            /* count available slots */
semaphore full = 0;             /* count occupied slots */

void producer(void)
{
        int item;
        while (1)
        {
                item = produce_item();
                down(&empty);
                down(&mutex);
                insert_item(item);
                up(&mutex);
                up(&full);
        }
}

void consumer(void)
{
        int item;
        while (1)
        {
                down(&full);
                down(&mutex);
                item = remove_item();
                up(&mutex);
                up(&empty);
                consume_item(item);
        }
}
```

Figure 3: Solving Producer and Consumer Problem with Semaphores.

```
; From Andrew Tanenbaum's Modern Operating Systems, 2nd Edition

enter_region:
         TSL RX, LOCK
         CMP RX, R0  ; assume R0 always contains ZERO
         JNE enter_region
         RET         ; return to caller once critical region is entered

leave_region:
         SW R0, MUTEX
         RET

================

; the following code can be used to make MUTEX for user level thread package
; From Andrew Tanenbaum's Modern Operating Systems, 2nd Edition

mutex_lock:
         TSL RX, MUTEX
         CMP RX, R0
         JZE ok
         CALL thread_yield
         JMP mutex_lock
ok:
         RET

mutex_unlock:
         SW R0,MUTEX
         RET
```

Figure 4: Examples of using TSL.

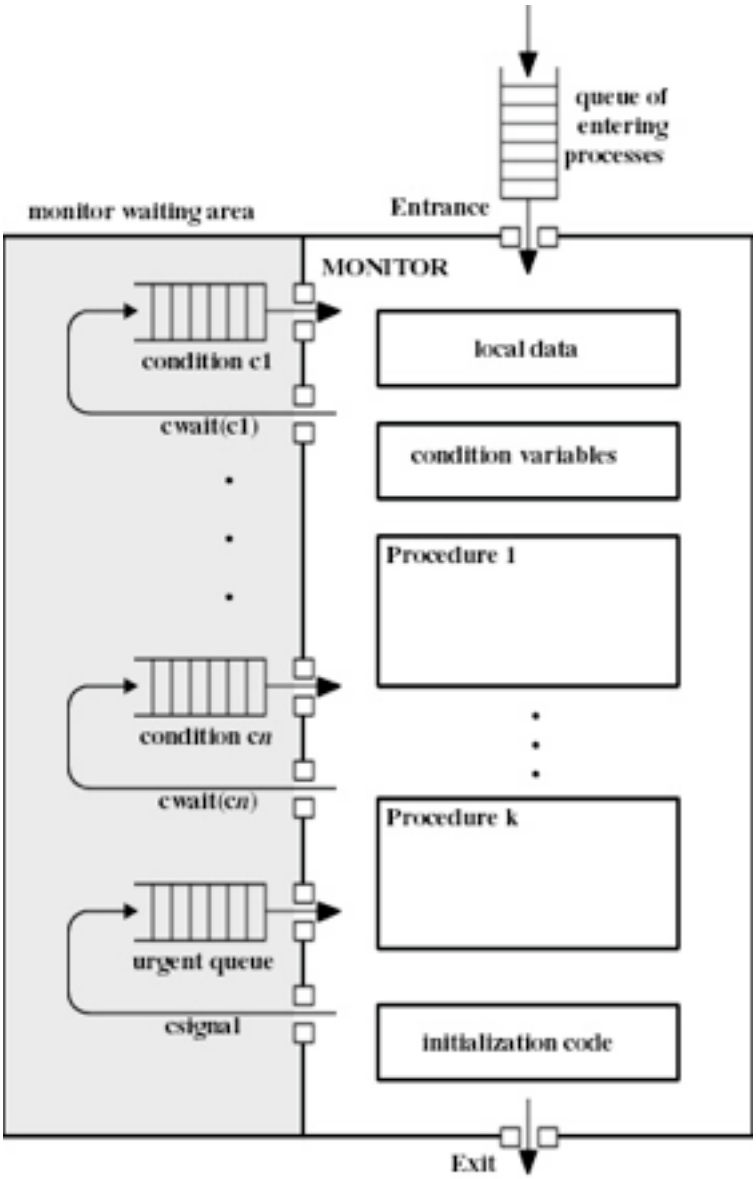
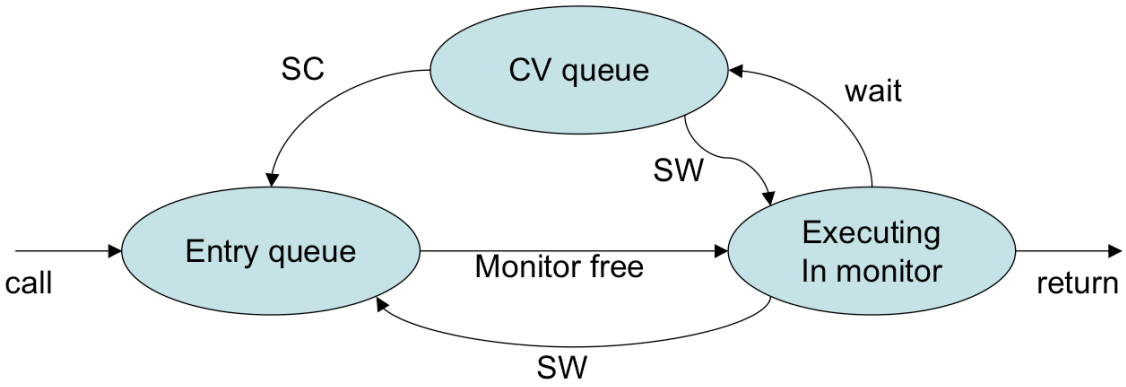


Figure 5: Monitor's Structure.



From G. Andrews, “Foundation of Multithreaded, Parallel, and Distributed Programming”, Addison Wesley, 2000

Figure 6: Two Signaling Discipline for Monitors.

```
char buffer [n]; int nextin, nextout; int count; int notfull, notempty;
nextin = 0; nextout = 0;  count = 0;

void append (char x)
{
        if (count ==n)
                cwait (notfull);
        buffer[nextin] = x;
        nextin = (nextin + 1) % n;
        count ++;
        csignal (notempty);
}

void producer ( )
char x;
{
        while (true)
        {
                produce (x);
                append (x);
        }
}
```

```
void take (char x)
{
        if (count == 0)
                cwait (notempty);
        x = buffer[nextout];
        nextout = (nextout + 1)%n;
        count --;
        csignal (notfull);
}

void consumer ( )
char x;
{
        while (true)
        {
                take (x);
                consume (x);
        }
}
```

Figure 7: A Solution to the Producer/Consumer Problem Using Monitor.