# CMSC 412 Midterm #1 (Spring 1996)

1.) (15 points) Define the following terms:

a) thread

code in execution, like a process but without as much state (esp without an address space)

b) policy vs. mechanism

policy - what to do (e.g., give a class of users a specific priority)

mechanism - how to carry out the policy (e.g. support different priorities for different processes)

c) preemptive multi-tasking

The ability to run more than one process at a time (though not simultaneously) and to have the operating system switch between tasks without the task voluntarily giving up the prcessor.

d) critical section (three conditions)

(1) mutual exclusion,
(2) tasks not in the CS or waiting to enter CS can't hold up tasks from entering the CS.
(3) bounded waiting time; other tasks can enter the CS a finite number of times before a waiting task gets in.

2.) (10 points) **Statement:** In DOS, user programs change the hardware interrupt vector.

a) If user programs can do this, why is it impossible to provide processor protection?

User program could change the timer interrupt and prevent the OS from getting control of the processor back.

b) Windows NT provides processor protection and can run DOS programs? Explain how this might be accomplished?

Virtualize the interrupt vector. Processes can change the virtual interrupt, but the hardware timer interrupt is still controlled by the kernel.

3.) (20 points) In a system that contains only one instance of each resource, circular waiting is a necessary and sufficient condition for deadlock.

a) Why is circular waiting sufficient if there is only one instance of each resource, but only necessary if there is one instance of each resource?

If there is more than one instance of a resource, it is possible for a circular waiting condition to be broken when a third party (not involved in the circular wait) releases a resource.

b) Consider a system with one instance of each resource, and the condition that requests for resources must be satisfied in the order in which they are received. Give an algorithm that takes as input requests to allocate and release resources and determines if the system is deadlocked.

maintain a list of all processes waiting for each resource (make it a queue to ensure resources are handed out FIFO). For each process, maintain a list of resources held.

```
Deadlock() {
    inLoop[1..nproc] of Boolean;

  for each process p {
      inloop = false;
       while (p <> null)  {
           if inLoop[p] then return true
           else
              inLoop[p] = true;
              p = waitOn[p];
       }
   }
}

while not done {
   if request is acquire
      if requested resource is free
          requested.resource.holder = requesting process;
       else
          enqueue(request.resource, requesting process);
           waitingOn[requesting process] = requested.resource.holder;
   else if request is release
       request.holder = dequeue(resource);
```

4.) (25 points) You have to solve a variation of the readers-writers problem, in which multiple writers can write at the same time. Specifically, there are readers and writers. Multiple reads at the same time are allowed. Multiple writes at the same time are allowed. A read and a write at the same time is not allowed. Provide a solution using semaphores with the following properties:

- no busy waiting.
- starvation-free (i.e. a continuous stream of readers does not starve writers, and vice versa) is desirable but not compulsory (**but you will lose some points**).
- you cannot use process ids and you cannot have a separate semaphore for every process.

Below is a skeleton program for you to build upon by supplying code for the boxes and perhaps introducing more variables. You are also welcome to disregard this skeleton and come up with something else.

| | |
|---|---|
| `nr_active`: integer initialized to 0. | /* number of reads currently executing */ |
| `nr_wait`: integer initialized to 0. | /* number of readers currently waiting */ |
| `r_sem`: semaphore initialized to 0. | /* readers wait here */ |
| `nw_active`: integer initialized to 0. | /* number of writes currently executing */ |
| `nw_wait`: integer initialized to 0. | /* number of writers currently waiting */ |
| `w_sem`: semaphore initialized to 0. | /* writers wait here */ |
| **mutex: semaphore initialized to 0.** | /* **protect shared object** */ |

Readers execute this code:

```
while (1) {
    P(mutex);
   if (nw_active + nw_waiting == 0)
         nr_active++;
         V(r_sem);
   else
         nr_waiting++;
   V(mutex);
   P(r_sem);

   Read operation;

   P(mutex);
   nr_active--;
   if (nr_active == 0) and (nw_waiting != 0)
         while (nw_waiting)
            nw_active++;
            nw_waiting--;
            v(w_sem);
   V(mutex);
}
```

Writers execute this code:

```
while (1) {
    P(mutex);
   if (nr_active + nr_waiting == 0)
         nw_active++;
         V(w_sem);
   else
         nw_waiting++;
   V(mutex);
   P(w_sem);

   Write operation;

   P(mutex);
   nw_active--;
   if (nw_active == 0) and (nr_waiting != 0)
         while (nr_waiting)
            nr_active++;
            nr_waiting--;
            v(r_sem);
   V(mutex);
}
```

5.) (15 points) Your friend, Lazy Student, wants to test his implementation of multiplication (shown below) as a process in your kernel from assignment number two. Why might Lazy not want to run multiply(5000, 2)? Explain (in as much detail as possible) what would happen if he tried this.

```
int multiply(int x, int y)
{
      if (x == 1) {
            return(y);
      } else {
            return(y + multiply(x-1,y));
      }
}
```

The stack can overflow which will overwrite memory (what memory is altered depends on how safe_malloc works and what else has been allocated in the heap).

6.) (15 points) Using test-and-set instructions, provide an algorithm that solves the dining philosophers problem. Recall that in this problem, n philosophers sit at a round table with one chop stick between each philosopher. Philosophers either are thinking or eating (rather dull people :). To eat, they need to acquire the chopsticks to their left and their right. Your solution must work for any number of philosophers and ensure that none of them starves to death.

```
// lock - only let one in at a time
lock() { while test-and-set(); }

// unlock - clear bit
unlock() { clear(); }

// waiting is an array of Booleans indicating who is waiting
// the queue ensure FIFO service
// solution only  lets one eat at a time
Phil {
   lock();
   if (!eating)
      eating = true;
   else
      waiting[my_id] = true;
      enqueue(my_id);
   unlock();
   while (waiting[my_id]);
   // eat

   lock();
   if (next = dequeue()) then
      waiting[next] = false;
   else
      eating = false;
   unlock();
```