CS-537: Midterm Exam (Spring 2001)

Please Read All Questions Carefully!

There are seven (7) total numbered pages

Name: _____

Grading Page

	Points	Total Possible
Part I: Short Answers		$(12 \times 5) \rightarrow 60$
Part II: Long Answers		$(2 \times 20) \rightarrow 40$
Total		100

Name: _____

Part I: Short Questions

The following questions require short answers. **Each of 12 is worth 5 points (60 total)**.

1. Which of the following are more like policies, and which are more like mechanisms? **For each answer, circle policy or mechanism.**

   (a) The timer interrupt Policy / Mechanism

   (b) How long a time quantum should be Policy / Mechanism

   (c) Saving the register state of a process Policy / Mechanism

   (d) Continuing to run the current process when a disk I/O interrupt occurs Policy / Mechanism

   The interrupt and register state saving are mechanisms (the "how"), whereas the quantum length and decision to run a particular process are clearly policies.

2. For a workload consisting of ten CPU-bound jobs of all the same length (each would run for 10 seconds in a dedicated environment), which policy would result in the **lowest average response time**? **Please circle ONE answer.**

   (a) Round-robin with a 100 millisecond quantum

   (b) Shortest Job First

   (c) Shortest Time to Completion First

   (d) Round-robin with a 1000 nanosecond quantum

   Response time is the time from the demand from service until the first service. Shortest-job first and shortest-time-to-completion will make some processes (long ones) wait arbitrarily long, so those can be ruled out. Thus, it is clearly one of the RRs. The one with the shorter time quantum wins (1000 ns is much less than 100 ms).

3. Assume we divide the heap into fixed-sized units of size S. Also assume that we never get a request for a unit larger than S. Thus, for any request you receive, you will return a single block of size S, if you have any free blocks. What kind of **fragmentation** can occur in this situation, and why is this bad?

   Internal fragmentation may occur. This is bad simply because it potentially wastes space, e.g., if all requests are for 1 byte, S-1 bytes will be wasted per request. It is called internal fragmentation because the waste is "internal" in the block that has been allocated.

4. Name and describe **two** examples of where we saw the OS needed support from the hardware in order to implement some needed functionality (**hint**: think about process management and memory management).

   Many possible answers. Base and bounds registers for memory management support. Test and set instruction for synchronization. Timer interrupt to re-gain control of the CPU.

5. What is a **cooperative** approach to scheduling, and why is it potentially a bad idea?

   Cooperative scheduling assumes that processes will voluntarily give up the CPU. It is potentially bad because a buggy or malicious process can commandeer the CPU and never relinquish it.

6. Assume we run the following code snippet. After waiting for a "long" time, how many processes will be running on the machine, ignoring all other processes except those involved with this code snippet? You can assume that fork() never fails. Feel free to add a short explanation to your answer.

```
void
runMe()
{
    for (int i = 0; i < 23; i++) {
        int rc = fork();
        if (rc == 0) {
            while (1) ;
        } else {
            while (1) ;
        }
    }
}
```

Number of processes running:

Two processes. The parent enters the code, forks a child, and then spins eternally. The child gets created, and starts spinning too. Thus, we have parent and child, each spinning forever (well, until you hit control-C).

7. Assume the following code snippet, where we have two semaphores, 'mutex' and 'signal':

```
Thread 1                                    Thread 2

mutex.P();                                  mutex.P();
if (x > 0)                                  x++;
    signal.V();                             signal.V();
mutex.V();                                  mutex.V();
signal.P();
```

We want 'mutex' to provide mutual exclusion among the two threads, and for 'signal' to provide a way for thread 2 to activate thread 1 when 'x' is greater than 0. What should the initial values of each of the two semaphores be? (Assume that 'x' is always positive or zero, and that there are only these two threads in the system).

Value of mutex: 1

Value of signal: 0

Mutex must be '1' to provide mutual exclusion. Signal must be '0' to ensure the proper signalling occurs (that is, thread 1 will only pass through the signal.P() if signal has been incrememted first via a V(), either by itself or thread 2).

8. Assume we manage a heap in a **best-fit** manner. What does best-fit management try to do? What kind of **fragmentation** can occur in this situation, and why is this bad?

Best-fit management finds the smallest block that will fulfill the current request (equal-to or greater-than). Organizing memory in such a manner can lead to external fragmentation. This is bad because a request that asks for S bytes of memory may be denied even when more than S bytes are free, but the S bytes are scattered throughout memory.

9. Which of the following will **NOT** guarantee that deadlock is avoided? **Please circle all that apply.**

   (a) Acquire all resources (locks) all at once, atomically

   (b) Use locks sparingly

   (c) Acquire resources (locks) in a fixed order

   (d) Be willing to release a held lock if another lock you want is held, and then try the whole thing over again

   Using locks sparingly does not do anything for you, and can still lead to deadlock.

10. Why is stack-based memory management usually much faster than heap-based memory management? If it's so much faster, why do we ever use the heap?

    The stack's advtantage is simplicity, where allocation is just a pointer increment, and deallocation is a decrement. Simplicity implies speed in this case. We use the heap for flexibility, e.g., for data structures that are not allocated/deallocated in a stack discipline.

11. For a workload consisting of ten CPU-bound jobs of varying lengths (half run for 1 second, and the other half for ten seconds), which policy would result in the lowest total run time **for the entire workload? Please circle all that apply.**

    (a) Shortest Job First

    (b) Shortest-Time to Completion First

    (c) Round-robin with a 100 millisecond quantum

    (d) Multi-level Feedback Queue

    Total time is not altered by scheduling policy, and thus all policies are equivalent. You could argue that SJF is the fastest, because of a lack of context switches. But then you would have to know the parameters of the MLFQ, which were not specified...

12. Assume we have a system that performs dynamic relocation of processes in physical memory. A process that has just started needs 2000 total bytes of memory in its address space. The OS currently has two free regions: the first between physical addresses of 1000 and 2000, and the second between physical addresses 5000 and 7000. What value would end up in the **base** register for this process, and what value would be in the **bounds** register? (write down any assumptions that you make about the bounds register)

    Value of base: 5000

    Value of bounds: 2000 or 7000

Part II: Longer Questions

The second half of the exam consists of two longer questions, **each worth 20 points (total 40)**.

1. **Monit-or Not, Here I Come.**

   You are stuck using some language that only provides **monitors** for mutual exclusion. However, you don't really like monitors all that much, and decide to implement a new class that looks a lot more like the **Binary semaphore** we all know and love. The monitor class looks something like this.

```
monitor class BinarySemaphore {
    // your class variables go here
    void P() {
        // code for P() goes here
    }
    void V() {
        // code for V() goes here
    }
}
```

   You have at your disposal a **condition** class, which has three methods: **signal()**, **broadcast()**, and **wait()**, which all function as you would expect given our class discussion. You are also expected to use **Hoare** semantics.

   a): What variables would you declare as part of this monitor class, and what would you initialize them to?

```
int count = 1;
condition c = new condition();
```

   b): Write the pseudo-code for your monitor implementation of P() here.

```
if (count == 0)
    c.wait();
count = 0;
```

   c): Write the pseudo-code for your monitor implementation of V() here.

```
count = 1;
c.signal();
```

   b: Assume you had to use **Mesa** semantics instead. What would have to change in your code? If you didn't change the code, what bad thing could happen?

   The 'if' in the P() code should be a 'while'. If this change isn't made, the code will not guarantee mutual exclusion (think about another process entering P() just as a waiting process is woken.

2. **Race to the Finish**

Assume we are in an environment with many threads running. Take the following C code snippet:

```
int z = 0; // global variable, shared among threads
void update (int x, int y) {
    z += x + y;
}
```

Assume that threads may all be calling update with different values for x and y.

a): Write assembly code that implements the function update(). Assume you have three instructions at your disposal: (1) **load [address], Rdest**, (2) **add Rdest, Rsrc1, Rsrc2**, and (3) **store Rsrc, [address]**. Also, feel free to assume that when update() is called, the value of 'x' is already in R1, and the value of 'y' is in R2.

```
add r2, r1, r2  // combine x and y
ld 'z', r3      // load z
add r2, r2, r3  // combine x, y, and z
store r2, 'z'   // store new value of z
```

b): Because this code is not guarded with a lock or other synchronization primitive, a "race condition" could occur. Describe what this means.

A race condition occurs when two processes 'race' to update a shared variable (in this case, 'z'). It is a race because the outcome is indeterminate; the outcome depends on how the scheduler interleaves the two threads.

c): Now, label places in the **assembly** code where a timer interrupt and switch to another thread could result in such a race condition occuring.

```
add r2, r1, r2  // combine x and y
ld 'z', R3      // load z
(SWITCH HERE IS A PROBLEM)
add r2, r2, r3  // combine x, y, and z
(SWITCH HERE IS A PROBLEM)
store r2, 'z'   // store new value of z
```

Basically, once 'z' is loaded, and before it is stored, we are in danger.

d): Now, assume we change the C code as follows:

```
void update (int x, int y) {
    z = x + y; // note we just set z equal to x+y (not additive)
}
```

If two threads call update() at "nearly" the same time, the first like this: 'update(3,4)', and second like this: 'update(10,20)', what are the possible outcomes? If we place a lock around the routine (e.g., before setting z = x + y, we acquire a lock, and after, we release it), does this change the behavior of this snippet?

No problem here, because 'z' is not incremented, simply over-written. Thus, whether there is mutual exclusion or not, the value of 'z' will either be '7' if thread 1 went last, or '30' if thread 2 went last.