COS 318: Midterm Exam (October 23, 2012)

(80 Minutes)

Name:

This exam is closed-book, closed-notes. 1 single-sided 8.5x11" sheet of notes is permitted.

No calculators, laptop, palmtop computers are allowed during the exam.

The points in the parenthesis at the beginning of each question indicate the points given to that question; there are 40 points in all.

Write all your answers directly on this paper. Use the blue book only if you run out of space. Make your answers as concise as possible.

Partial credit will be given if you can show your work in arriving at solutions.

1	
2	
3	
4	
5	
Total	

Pledge (please write out "I pledge my honor that I have not violated the Honor Code during this examination" and then sign):

1 Short Questions (10 points)

Answer the following questions. Use exactly **one or two** sentences to describe why you choose your answer. Without the reasoning, you will not receive any points.

a. (1 point) Intel Pentium processors have special instructions such as SSE for multimedia processing. These instructions have their own registers. Are the registers for such instructions part of the context of a process?

Yes. This is because any programs can use such instructions at any time.

b. (1 point) Are the registers for I/O devices part of the context of a process?

No. I/O devices are accessed via device driver calls in the kernel.

c. (2 point) Can Monitors deadlock?

Yes. A program inside a monitor can hold a resource and request for another while another program can do similarly to form a circular chain.

d. (2 point) Provide two advantages of threads over multiple processes.

Lower overhead of context switch.

Easier to share data.

e. (2 points) Many operating systems designed to run only on uniproccessors use disabling of interrupts to create critical sections in the code. Explain briefly how the interrupt disabling prevents multiple threads from entering the critical section.

Suppose two threads can enter a critical section simultaneously on a uniprocessor. Thread A gets into the critical section first. Then the scheduler suspends thread A and run thread B. Thread B gets into the critical section. This is not possible because the scheduler can only be involved either by calling Yield() or a timer interrupt. Thread A cannot call Yield() because it is in the critical section. A timer interrupt is not possible because interrupts are disabled.

f. (2 points) Explain briefly why implementing critical sections by disabling interrupts will not work on a multiprocessor.

Disabling interrupts will not work on a multiprocessor because disabling interrupts only happens on the CPU it runs. Another thread can run on another CPU and gets into the critical section.

2 CPU Scheduling (10 points)

a) (6 points) Suppose that the following processes arrive for execution at the times indicated. Each process will run the listed amount of time. In answering the questions, use non-preemptive scheduling and base all decisions on the information you have at the time when the decision must be made.

Process	Arrival Time (sec)	Running Time (sec)
P_1	0.0	16
P_2	0.3	5
P_3	1.0	3

- (2 points) What is the average turnaround time for these processes with the FCFS scheduling algorithm?

  (16 + (16 – 0.3 + 5) + (21 – 1 + 3)) / 3 = 19.9

- (2 points) What is the average turnaround time for these processes with the STCF scheduling algorithm?

  (16 + (16 – 1 + 3) + (19 – 0.3 + 5)) / 3 = 19.23

- (2 points) The STCF algorithm is supposed to improve performance, but notice that we chose to run process P_1 at time 0.0 second because we did not know that shorter processes would arrive soon. Compute what the average turnaround time will be if the CPU is left idle for the first 1 second and then STCF is used.

  (3 + (4 – 0.3 + 5) + (9 + 16)) / 3 = 12.23

b) (2 points) Suppose that there are *n* processes in the ready queue to be scheduled on one processor. How many possible different schedules are there? Give a formula in terms of *n*.

n (n-1) … (1) = n!

c) (2 points) Describe how a lottery scheduling algorithm could be made to approximate a non-preemptive CPU scheduling algorithm that achieves the shortest average turnaround time.

To approximate SRTF, give short running jobs more tickets and long running jobs fewer tickets.

3 Input and Output (5 points)

Briefly list the steps of a blocked write system call in a typical monolithic operating system (such as Unix) by a process. This system call uses DMA hardware to write to a disk device.

- A process issues a write call which executes a system call
- System call code checks for correctness
- Make a device driver write call
- Device driver allocates a buffer and copy data from user buffer to kernel buffer
- Initiate a DMA write data transfer
- Block the current process and schedule a ready process
- Device generates an interrupt upon completion of the DMA transaction
- Interrupt handler makes the blocked process ready
- Scheduler wakes up blocked process at some point
- The write process continues when it is scheduled to run

4 Synchronization design (5 points)

Your colleague has designed a synchronization mechanism for editing multiple files. The goal is to maintain consistency when a user applies edits to a number of files simultaneously. Your colleague designed **`MultiLock()`** to be called once before an editing session and **`MultiUnlock()`** to be called once with the same arguments after the session:

```
MultiLock( File *fileList[], int n ) {
    int i;

    for (i = 0; i < n; i++ )
        Acquire( fileList[i]->lock );
}
```

```
MultiUnlock( File *fileList[], int n ) {
    int i;

    for (i = 0; i < n; i++ )
        Release( fileList[i]->lock );
}
```

where **File** is a structure that has **lock** as one of the field.

Your job is to review the design, figure out whether there are any issues with **MultiLock()** and **MultiUnlock().** If there are, please provide examples to show the issues first, and then provide pseudo-code to fix the issues.

The following condition may cause a deadlock. User A calls:

```
fileList[0] = file1; fileList[1] = file2;
MultiLock( fileList, 2 );
```

After user A successfully acquired the lock of **file1** but before the lock of **file2**, user B calls:

```
fileList[0] = file2; fileList[1] = file1;
MultiLock( fileList, 2 );
```

The following fix the problem mentioned above:

```
MultiLock( fileList[], int n ) {
    int i; FILE sortedList[MAX];

    Sort( fileList, sortedList, n );
    for (i = 0; i < n; i++ )
        Acquire( sortedList[i].lock );
}
```

The primitive **Sort()** sorts **fileList** according to the value of each file descriptor and stores the results to **sortedList**. There is no need to modify **MultiUnlock()**.

5 Monitors (10 points)

At a job interview, you are asked to show how to implement an eventcount package with Mesa-style monitor. An eventcount is an object that keeps a count of the number of events that have occurred in a system. There are four primitives of the eventcount abstraction:

- **Init(EC ec)**: Initialize the eventcount object.
- **Advance(EC ec)**: Increment the counter in the eventcount object by 1.
- **Read(EC ec)**: Return the value of the counter in the eventcount object.

- **`Await( EC ec, int v ):`** Block until the value of the counter in the eventcount object reaches a specific value `v`.

Note that `Advance`, `Read` and `Await` primitives should be atomic. User programs can use this set of primitives to implement synchronizations based on event occurrences conveniently without using other synchronization primitives.

Your job is to first define the data structure of the eventcount and then show how to use Mesa-style monitor to implement the primitives.

The data structure definition for EC is:

```
struct EC {
  int counter;
  Mutex lock;     /* mutex */
  Cond c;         /* condition variable */
};
```

The new code is:

```
Init( EC ec ) {
  ec.counter = 0;
  Cond_Init( ec.c );
  Mutex_Init( ec.lock );
};

Read( EC ec ) {
  int r;
  Acquire( ec.lock );
  r = ec.counter;
  Release( ec.lock );
  return r;
};

Advance(EC ec) {
  Acquire( ec.lock );
  ec.counter++;
  Release( ec.lock );
  Broadcast( ec.c );
};

Await( EC ec, int v ) {
  Acquire( ec.lock );
  while ( ec.counter < v )
    Wait( ec.lock, ec.c );
  Release( ec.lock );
};
```