Operating Systems
WPI – CS3013
February 5, 20067
Midterm Exam

Name _____

Answer the following **SIX** questions. There's lots of partial credit, so put down what you know. You are to use **NO** notes or papers for this exam.

Problem 1: Short Answers: (15 Points)

a) What is a virtual machine?
A virtual machine is software that acts as if it's hardware. It traps a hardware request made by an OS and emulates the behavior desired by that request In the case of a Java Virtual Machine, it allows a common program to run on any hardware – it provides the middleware required to run Java bytecode anywhere..

b) What is Little's Law? What does it mean?
Little's Law states that the Residence time of a request is equal to the Number of Requests waiting in a queue times the average service time of that request.
*R = N * S.*

c) What is the job of a dispatcher?
In a scheduler, the dispatcher has the task of removing a process (as represented by its PCB) from the ready queue and running that process on the processor. In order to run this new process, the dispatcher must save the context / current state of the process that's already running.

Problem 2: Scheduling (25 Points)

A number of jobs enter a system with characteristics shown below.

job	arrival time	required processing time
1	0	3
2	1	5
3	3	6
4	5	2
5	7	1

Answer this problem by filling in the Tables.

Determine the **average** residence (or completion) time for these jobs using each of these scheduling mechanisms:

a) **FCFS** **AVERAGE RESIDENCE TIME =** _____***8.4***_____

I've started the solution here.

Time	**Arrival**	**Start**	**Finish**
0	Job 1	Job 1	
1	Job 2		
3	Job 3	*Job 2*	Job 1
5	*Job 4*		
7	*Job 5*		
8		*Job 3*	*Job 2*
14		*Job 4*	*Job 3*
16		*Job 5*	*Job 4*
17			*Job 5*
R = ((3 – 0) + (8 – 1) + (14 - 3) + (16 – 5) + (17 – 7)) / 5			
= (3 + 7 + 11 + 11 + 10) / 5 = 42 / 5 = 8.4			

b) Shortest job first with preemption.

AVERAGE RESIDENCE TIME = _____**5.8**_____

Time	**Arrival**	**Start**	**Finish**
0	*Job 1*	*Job 1*	
1	*Job 2*		
3	*Job 3*	*Job 2*	*Job 1*
5	*Job 4*	*Job 4*	*(2 preempted)*
7	*Job 5*	*Job 5*	*Job 4*
8		*Job 2*	*Job 5*
11		*Job 3*	*Job 2*
17			*Job 3*
R = ((3 – 0) + (11 – 1) + (17 - 3) + (7 – 5) + (8 – 7)) / 5			
= (2 + 10 + 14 + 2 + 1) / 5 = 29 / 5 = 5.8			

Problem 2: - Continued

c) Round robin with time quantum of 2 seconds (no preemption).

AVERAGE RESIDENCE TIME = _____

Time	Arrival	Start	Finish
0	*Job 1*	*Job 1*	
1	*Job 2*		
2		*Job 2*	
3	*Job 3*		
4		*Job 3*	
5	*Job 4*		
6		*Job 4*	
7	*Job 5*		
8		*Job 5*	*Job 4*
9		*Job 1*	*Job 5*
10		*Job 2*	*Job 1*
12		*Job 3*	
14		*Job 2*	
15		*Job 3*	*Job 2*
17			*Job 3*
R = ((10 – 0) + (15 – 1) + (17 - 3) + (8 – 5) + (9 – 7)) / 5			
= (10 + 14 + 14 + 3 + 2) / 5 = 43 / 5 = 8.6			

Problem 3: Threads and Processes: (15 Points)

```
OUTPUT 1:
The adder value is = 0
The subtracter value is = 1
The adder value is = 0
The subtracter value is = 1
The adder value is = 0
The subtracter value is = 1
The adder value is = 0
The subtracter value is = 0
The adder value is = 0
The subtracter value is = 0

OUTPUT 2:
The subtracter value is = -1
The adder value is = 1
The subtracter value is = -2
The adder value is = 2
The subtracter value is = -3
The adder value is = 3
The subtracter value is = -4
The adder value is = 4
The subtracter value is = -5
The adder value is = 5
```

In class we ran a program that, depending on the command line option gave one or the other of the outputs shown here.

What's different about these two environments?

In the first instance, we ran the program and created threads. In the second instance, we created separate processes by using a fork command.

Why do the two executions produce different results?

Threads in the same process share memory – so the two threads are both working on the same global variable, one incrementing and one decrementing.
When a fork occurs, the memory is shared as long as neither process WRITES to the memory (they can read shared values.) But when either process writes to the memory, then each process gets its own copy of the memory – thus they do not see the same memory.

Problem 4: Critical Section (15 Points)

The following code claims to provide a solution to the critical section problem for two processes. Explain how the code **does** or **does not** meet the **THREE** requirements for a critical section.

```
Shared Boolean  Flag[2];        // Initially Flag[0] and Flag[1] = FALSE;
                                // When flag == TRUE, it indicates a user wants to
                                // get into the critical section.
```

For Process 0:

```
while( TRUE )
{
        Flag[0] := true;
        while ( flag[1] ) ;
        // critical section
        flag [0] = false;
        remainder section
}
```

For Process 1:

```
while( TRUE )
{
        flag[1] := true;
        while (flag[0]);
        // critical section
        flag [1] = false;
        remainder section
}
```

Does this code satisfy the THREE criteria for a critical section?? Explain why or why not.

Here are the definitions:
Mutual Exclusion: *No more than one process can execute in its critical section at one time.*
Progress: *If no one is in the critical section and someone wants in, then those processes not in their remainder section must be able to decide in a finite time who should go in.*
Bounded Wait: *All requesters must eventually be let into the critical section.*

The issue occurs when P0 sets its flag to True, P1 sets its flag to True, then P0 checks the flag of P1 and finds it can not proceed, then P1 checks the flag of P0 and finds it can't proceed either.

Mutual Exclusion – OK – only one process can get into the critical section.
Progress – Violated – the two processes can't decide who should go first.
Bounded Wait – This is not the primary criteria violated here. You'll get partial credit if you claim this one.

Problem 5: Scheduling (10 Points)

General purpose operating systems handle a mix of processes using short and long amounts of CPU.

(a) What is the difference between a long and short process in terms of process scheduling? Give an example of each type of process.

A long process is one where the CPU must be grabbed back by the scheduler/interrupt handler after it's time quantum is exhausted. A short process is one that gives up the CPU voluntarily. Generally, interactive programs with a user sitting at a terminal are short processes. Long processes often run in the background and have little user or IO expectations.

(b) Describe how the Linux credit-based process scheduling policy accounts for both short and long processes.

Linux gives a "credit" to processes who voluntarily give up the CPU. This credit is in the form of an improved priority that means the short processes are placed ahead of the long processes on the ready queue and thus are given preferential treatment by the dispatcher.

Problem 6: The Big Picture: (20 Points)

On the following page, you see an overview schematic of how an operating system is constructed in order to perform system calls and scheduling.

On that picture are a number of boxes. These boxes actually correspond to the picture I showed in class. I don't expect you to remember the exact names I used for the functions, but please write a brief statement of the tasks that need to occur in each box in order to accomplish a disk read. Plan your answers so that you provide an overview of the operations that must occur to do the disk read. HINT: The functions are lettered in approximate chronological order.

Function A: *There's a function that receives ALL system calls as they come into the kernel. The task of this function is to validate the system call in general, to figure out what system call is being requested, and then pass the request off to the function that will actually do the work.*

Function B: *This is a function that does the specific work of the **DiskRead**(). It will validate the parameters, and verify that the memory needed for the function exists. It then puts the request (or the process PCB) on the Disk Queue (a), starts the disk doing the request if the disk isn't already running, and calls the scheduler to run someone else.*

Function C: *This routine is essentially "**give up the CPU**". Its job is to unload the current process from the processor, and to save away the state of this current process in its PCB.*

Function D: *This is the **dispatcher** whose job it is to get a process from the Ready Queue (b) and put it on the processor. This is NOT a decision making function, but merely takes the process on the front of the queue and runs it.*

Function E: *This is the **interrupt handler**. All requests by the hardware for OS service come here. The Interrupt Handler then determines the type of interrupt and goes to the specific handler. In our example, the request is passed off to the Disk Handler.*

Function F: *The **disk interrupt handler** removes the first process on the disk queue (since that processes disk operation is now complete) and determines if that process is ready to go back on the ready queue. It then finds out if there is another disk request already on the Disk Queue that should be serviced. If so, then it asks the disk to do that request.*

Function G: ***MakeReadyToRun** accepts the PCB and puts it on the Ready Queue (b). This routine has the intelligence to determine where on the Queue to put this process. The position is typically determined by the priority of the process.*

Queue a: *This is the **Disk Queue**. Items are placed on here in priority order – what's first on the list will be executed next. Processes on this queue are in a Waiting State.*

Queue b: *This is the **Ready Queue**. Items are placed on this list in priority order, with the most favorable priority at the front of the queue. This is typically implemented as a doubly linked list.*