1. (10 points) What is the purpose of system calls, and how do system calls differ from procedure calls?

2. (10 points) Define the difference between non-preemptive and preemptive scheduling. Explain why preemptive scheduling is likely to be used in a multi-user computer system.

3. (10 points) Define turnaround time and waiting time. Give a mathematical equation for calculating both turnaround time and waiting time using a processes start time, $S_i$, finish time, $F_i$, and computation time $C_i$. The computation time, $C_i$, is the total time the process spends in the running state.

4. (15 points) An interactive computer system has enough memory to hold four programs in main memory. Assume programs spend 50% of their execution time waiting for I/O operations to complete. What fraction of time is the CPU idle? (Your answer here will depend on a number of factors such as the scheduling policy employed. To answer this question try and determine under what conditions the CPU would as busy as possible and as idle as possible. In all cases you should always assume that the operating system always tries to overlap I/O and computation whenever possible.)

5. (15 points) Round robin schedulers normally maintain a list of all ready processes, with each process occurring exactly once in the list.

What would happen if a process occurred twice in the list? What would be the major advantages and disadvantages of this scheme?

6. (15 points) Five jobs are waiting to be run. Their expected run times are 7, 2, 4, 12, and $X$. In what order should they be run to minimize average response time? (Your answer will depend on $X$.)
7. (20 points) Consider the following set of processes, with the length of the CPU burst time given in milliseconds:

<table>
<thead>
<tr>
<th>Process</th>
<th>Execution Time</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>45</td>
<td>2</td>
</tr>
<tr>
<td>$P_2$</td>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>$P_3$</td>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>$P_4$</td>
<td>30</td>
<td>5</td>
</tr>
<tr>
<td>$P_5$</td>
<td>40</td>
<td>4</td>
</tr>
</tbody>
</table>

The processes are assumed to have arrived in the order $P_1, P_2, P_3, P_4, P_5$, all at time 0. Consider the following scheduling policies:

- FCFS,
- SJF,
- a non-preemptive priority (a smaller priority number implies a higher priority),
- RR (with quantum = 5).

What is the turnaround time of each process for each of the scheduling algorithms? What is the waiting time of each process for each of the scheduling algorithms? Which of the schedules produced by each algorithm results in the minimal average waiting time (over all processes)?

8. (15 points) Consider the following preemptive priority scheduling algorithm based on dynamically changing priorities. Smaller priority numbers imply higher priority. When a process is waiting for the CPU (in the ready queue, but not running), its priority changes at a rate $\alpha$; when it is running, its priority changes at a rate $\beta$. All processes are given a priority of 10 when they are created (enter the ready queue for the first time). The parameters $\alpha$ and $\beta$ can be set to give many different scheduling algorithms.

a) What is the algorithm that results from $\beta > \alpha > 0$?

b) What is the algorithm that results from $\alpha < \beta < 0$?

c) What is the algorithm that results from $\alpha > \beta > 0$?

9. (20 points) Nachos is an instructional operating system that runs in a normal Unix process. Nachos will be the basis for your second and third programming projects. The purpose of these homework problems is to introduce you to the Nachos code now, before you have to make modifications to it. As such, the more you learn about Nachos now, the better off you will be later. Just as the first programming project introduced you to several new challenges, these homework problems and the last two projects will be quite challenging. Allow yourself plenty of time to complete them (in other words, start early!).

Begin by downloading your own copy of the Nachos archive. Then, follow the instructions in Building and Using Nachos.
Several documents have been created to introduce you to the Nachos system.

- **Introduction to Nachos**
- **Threads in Nachos**
- **System Calls in Nachos** (this is a very important document for these two problems)
- **Using SmartGDB**

In this problem you will trace the execution of the *Fork* system call (many of the system calls in Nachos have the same name as their Unix counterpart with the first letter capitalized). The purpose of this exercise is to understand the control-flow in a Nachos system call. A subsidiary goal is to understand the finer points of an implementation of the *Fork* call.

There are two ways to trace the control flow in Nachos. First, you could read through the source code and note the major function calls, etc. The Nachos source is neat and well commented. As such, this is not a bad way to begin.

The second way is to use a debugger to step through the execution of the operating system. For the purposes of this assignment, we have provided a pre-compiled version of the user program below. You can use the SmartGDB debugger to step through Nachos while running it. This way involves more initial work in learning the debugger, but may assist in getting a general idea of what's going on faster than diving right into the code.

Given the following Nachos user program:

```c
int main( ) {
    int foo;
    Fork();
    foo = 1;
}
```

Trace the execution of the program through the Nachos system from when it begins to execute the *Fork* call to the return of the main procedure.

Include the answers to the following questions with your trace:

a) What do the MIPS registers look like upon entry into the Nachos kernel?

b) When is a new process actually created? What are the steps involved in creating it?

c) Where does the new process start execution inside the kernel? Inside user space?

d) What functions are responsible for
   - Setting up the user (MIPS) registers of the new process after a fork?
   - Setting up the address spaces after the fork?

e) What is the relationship of the two processes’ memory spaces after the fork?

f) How does the assignment of foo change the relationship between the memory spaces of the two threads after the fork?
10. (20 points) The point of this question is to make sure you really understand the concept of context switching. Write down the sequence of procedure calls and returns, along with changes to the ready list, that would occur in Nachos if the program below were to execute. For each procedure call/return, give the name of the thread doing the call/return, the full procedure name, and the values of any significant arguments. Assume the initial thread is called Thread1 and the thread that is forked is called Thread2.

For this question you only have to list calls and returns to major routines defined in thread.cc, scheduler.cc, switch.s, exception.cc, mipssim.cc, machine.cc, and systemcall.cc along with calls to Interrupt::SetLevel(). In other words, you can ignore ASSERTs, DEBUG statements, calls to list routines, calls made internal to the interrupt simulation, any trivial inline functions, etc.

For example, your solution might begin:

RaiseException: Thread1 has raised an exception (SyscallException)

ExceptionHandler: Thread1 doing a system call (SyscallException)

do_system_call: Thread1 entering switch (SC_Fork)

...

You should assume for the purposes of this exercise that the following are true:

a) Interrupts are disabled (i.e. there is NO preemption). This is accomplished by using the '-noswitch' option when running Nachos. **If you trace the actual execution of Nachos, make sure you add the '-noswitch' option when starting Nachos**

b) The ready queue is empty except for the threads running the user code below.

It is acceptable (even encouraged) to generate your solution automatically (either by code modifications or by SmartGDB scripts).

```c
int main( ) {
    Fork( );
    Yield( );
}
```