Operating Systems Midterm

1. (10 points) A thread contains the thread status and stack. Explain (briefly) what is contained in the status and the stack. Answer: The stack contains data that is private to the thread. The status contains propterties which the OS needs for the management of the thread, such as the address of the next instruction, whether the thread is blocked or not, which resource the thread is waiting for if it is blocked.(p.53)

2. (10 points) What is the process descriptor? What is its use? Answer: A data structure which the OS keeps in order to manage a process. It contains the information necessary to manage the process. It contains the name of the process, a list of child processes, address space, resources, process ID (internal to the OS), process owner (i.e. login name).... (p.212).

3. (10 points) Upon the execution of a fork() command in Unix, a child process is created. Do the child and parent process share the same address space or do they share different address spaces? The child process contains copies of items belonging to the parent process. What are these items? Answer: The child and parent process occupy separate address spaces. The child process contains copies of the parents program text, data, and stack segments (p. 60).

4. (10 points) What is a trop instructionr? Explain what happens when a thread makes a system call, including the role of the trap table. Answer: The trap instruction is used to switch the the CPU mode from user to supervisor. It is used to run trusted (i.e. secure/safe) system code. It is a way to invoke system services (such as fork) safely, by switching to secure code which cannot be tampered with and is reliable.

5. (10 points) Starvation is a potential flaw in the shortest seek time first disk access algorithm. Explain how the scan algorithm attempts to solve this problem. What is the weakness of the scan algorithm? Answer: Scan goes from the "inside" edge of the disk to the "outside" and then reverses direction. This way it will reach all of the requests which arrive before it changes direction (and those requests which arrive for a track "in front of it" as well). Consequently, it does not suffer from the following problem: almost all of the ruquests come in for a set of tracks which are relatively close to one another, while some other requests, which are at a distance from the main group, do not receive service, i.e. they starve.

6. (10 points) Explain the difference in the way an OS would support user space threads and the way an OS would support kernel threads. Which approach does Unix follow? Answer: In user threads, the underlying OS actually supports processes, while a thread "package" such as Mach C threads function to multiplex threads on top of the process. In kernel based threads, the underlying OS does the multiplexing of the threads (e.g the thread scheduling for the CPU). Unix supports user threads via packages such as Mach C.(p.201).

7. (10 points) Consider a priority based CPU scheduling algorithm with fixed priorities. Explain how starvation can occur. Describe a solution to this problem. Can starvation occur in round robin scheduling? If so, suggest a solution. Answer: If we have fixed priorities, it is possible for higher priority threads/processes to dominate the CPU. A solution to this problem is to let the priority

of a thread/process increase with time. This way, a lower priority process will not have to wait forever, i.e. it will not starve. Round robin scheduling favors shorter jobs. Longer jobs might take a very long time to finish as a result of this. A way to improve this situation is to use a multi-level queueing structure for the processes, in which each level corresponds to a differenct priority. The RR mechanism could be used within each level. Both Unix and Windows make use of this approach.(p. 274)

8. (30 points) Two processes, p1 and p2 have been designed so that p2 prints a byte stream produced by p1. Write a skeleton for the proceedures executed by p1 and p2 to illustrate how they synchronize with one another using P and V. Answer: This problem is a variant of the producer-sonsumer problem:

```
producer() {
while(TRUE) {
b=produceByte(...);
P(mutex);
putByte(b);
V(mutex);
V(byteCount);
}
}

consumer()  {
while(TRUE)  {
P(byteCount);
P(mutex);
takeByute(b);
V(mutex);
}
}

semaphore mutex=1, byteCount=0;
fork(producer, 0);
fork(consumer, 0);
```