CS4410 - Fall 2008
Assignment 1 Solution

Q1. Give three examples of an explicit hardware mechanism that is motivated by specific OS services.

Answer:
Atomic operations for synchronization.
Kernel/user mode, base/limit registers, protected instructions for various forms of protection.
Interrupt vectors for handling interrupts.
Traps and trap vectors for handling internal errors and system calls.
Interrupts and memory-mapped communication for I/O

Q2. Can a process transition from waiting for an I/O operation to the terminated state? Why or why not?

Answer:
No. A process waiting for I/O must first transition to the ready queue and then to the running state before it may terminate.

Q3. What are the differences between user-level and kernel-level threads? Under what circumstances is one type better than the other? What is the essential cause of the difference in cost between a context switch for kernel-level threads and a switch that occurs between user-level threads?

Answer:
User-level threads are threads that the OS is not aware of. They exist entirely within a process, and are scheduled to run within that process's timeslices.
The OS is aware of kernel-level threads. Kernel threads are scheduled by the OS's scheduling algorithm, and require a "lightweight" context switch to switch between (that is, registers, PC, and SP must be changed, but the memory context remains the same among kernel threads in the same process).

User-level threads are much faster to switch between, as there is no context switch; further, a problem-domain-dependent algorithm can be used to schedule among them. CPU-bound tasks with interdependent computations, or a task that will switch among threads often, might best be handled by user-level threads.
Kernel-level threads are scheduled by the OS, and each thread can be granted its own timeslices by the scheduling algorithm. The kernel scheduler can thus make intelligent decisions among threads, and avoid scheduling processes which consist of entirely idle threads (or I/O bound threads). A task that has multiple threads that are I/O bound, or that has many threads (and thus will benefit from the additional timeslices that kernel threads will receive) might best be handled by kernel threads.

Kernel-level threads require a system call for the switch to occur; user-level threads do not.

Q4. Suppose that a process scheduling algorithm favors those processes that have used the least processor time in the recent past. Why will this algorithm favor I/O-bound processes, but not starve CPU-bound processes?

Answer:
I/O bound processes generally spend most of their time in the wait queue while an I/O operation is being performed. When their I/O finally completes, they will generally not execute for a long time before they initiate the next I/O operation.
Therefore, they will be favored in scheduling relative to long-running processes.
However, if CPU-bound processes are blocked long enough by I/O-bound processes, they too will eventually not have run for a long time and they will be scheduled.

Q5. Define makespan as the total time to complete a set of jobs. An Operating System wants to minimize its makespan for a given set of jobs. Imagine our Operating System has only 2 jobs: A and B. Provide a scenario where running the jobs sequentially will provide better performance (measured by having a smaller makespan) compared to running them in parallel. If such a scenario does not exist, explain why. Otherwise, explain the particulars of jobs A and B and how it performs better in the sequential environment.

Answer:
Imagine that both A and B are compute intensive jobs. In this case, they will not need to frequently make system calls or wait for I/O to complete. Running them sequentially will provide better performance because the overhead of context switching is eliminated.

Q6. Explain why system calls are needed to set up shared memory between two processes. Does sharing memory between multiple threads of the same process also require system calls to be set up?

Answer:
Because each process has its own address space, it needs to involve the kernel when dealing with other processes' address space. The kernel (not the process) has knowledge of the physical memory mapping of all processes so it can determine a chuck of memory that can be used to share among multiple processes.
Threads do not need to use a system call to share memory because, by definition, share their address space with other threads (of the same process, of course).

Q7. Suppose that we have a single-core, uniprocessor system which supports multiprogramming. At any given time, how many processes can be in the running state in this system?

Answer: One.