Midterm Examination
CS 111
MS Online Program
Spring 2013

Answer all questions. All questions are equally weighted. This is a closed book, closed notes test. You may not use electronic equipment to take the test.

1. What is the difference between partitionable resources, sharable resources, and serially sharable resources? Give an example of each kind of resource that might be found in a computer system.

   Partitionable resources can be divided into discrete parts, each of which can be given to a separate thread. Memory is a good example. Sharable resources can be used simultaneously by multiple threads. A dynamic library or a file are examples. Serially sharable resources can be used by multiple threads, but only one at a time. A network card is an example.

2. In a typical system, if there is a critical resource, it is difficult to maintain its utilization at a high level. Why? How would this apply to achieving high utilization in using a computer's single processor?

There is overhead associated with scheduling a shared resource. The more popular the resource, the more of its time is wasted due to the overhead. The most common techniques used to achieve high utilization are to minimize the number of requests for the resource and reduce the overhead associated with offering the resource to a using entity. For a processor, the former is achieved by giving processes relatively long time slices.

3. What is meant by separating policy and mechanism in the OS context? Why is it a good idea? Give an example of somewhere in an operating system where such separation provides benefits, and describe why it provides such benefits.

Policy describes what one wants to do. Mechanism describes how to do it. Separating policy from mechanism means you design mechanisms capable of implementing a wide range of policies. It's a good idea because it does not tie the algorithms and code of your OS to a single policy. One place it can provide benefits is in scheduling, where a scheduler can change time slices, work with or without priorities, and use one or several scheduling classes to achieve different scheduling effects (like high throughput vs. real time scheduling) without changing its basic code.

4. What is an interpreter's repertoire? For a process instance of the interpreter abstraction (for example, a compiler process), what determines the repertoire of the particular process?

The repertoire is the set of things the interpreter can do. For a process, the repertoire is determined by the code that the process runs.

5. Traps can occur in either user mode or supervisor mode. What is similar about traps that occur in these two modes? What is different?

The mechanics of handling the two kinds of traps are nearly identical. In both cases, the PC and PS are saved on the stack, the same first level handler is used, and the same mechanisms for saving and restoring registers and transferring control back to the interrupted code are used. The differences are that, for a trap in supervisor mode, the PS that is saved shows that the interrupted code was in supervisor mode, and that a second level handler will be chosen, in part, based on knowledge that the trap occurred in supervisor mode.

6. If you wish to go from providing soft modularity to hard modularity on a single machine, what must the operating system ensure happens? Describe three resources on a typical computer where the operating system must do something to ensure hard modularity, and briefly describe what sort of thing must be done.

The operating system must ensure that no thread can inadvertently or intentionally cross the hard modularity boundaries. 1). Memory: the OS must ensure that memory areas belonging to one thread cannot be accessed by another. 2). The file system: the OS must ensure that a process cannot read or write files for which it does not have access permission. 3). Scheduling: the OS must ensure that a thread cannot continue to run when its time slice has expired.

7. What is the relationship between interrupts and receive livelock?

Receive livelock occurs when requests arrive at a resource faster than they can be handled. Each receipt causes an interrupt. While that interrupt is being processed (and further interrupts are masked), another request arrives. As soon as the first interrupt's handling is done and interrupts are re-enabled, the second interrupt occurs, before any actual handling of the request is possible. In essence, in receive livelock, the processor only services interrupts and has no time to do actual request processing.

8. In the scheduling context, what is starvation? Why does it happen? What can be done to avoid it?

Starvation happens when the scheduler consistently fails to schedule some thread to execute, resulting in the thread waiting indefinitely. It happens when the scheduler uses a scheduling criteria that allows other threads to receive higher priority use of the processor. Examples would be explicit priorities (low priority processes might get starved if high priority processes keep arriving) or shortest job first (long jobs might get starved if short jobs keep arriving). It can be avoided either by using a scheduling method that guarantees that all processes get a turn (e.g., round robin), or by ensuring that all processes are promoted in the scheduling criterion dependent on how long it has been since they last ran.

9. Generally describe the sleep/wakeup race. What might happen as a result of such a race? How can it be prevented?

The sleep/wakeup race occurs when a thread tries to lock a resource, determines that the resource lock is held by someone else, and enters a sleep state. However, before the process is put on the queue of threads waiting for the lock, the sleeping thread hits a context switch. A second thread that holds the lock then releases it and looks to see if any thread is on the queue of threads waiting for the lock. Seeing no one there, this thread merely releases the lock and doesn't wake up the first process. The sleeping thread might never be woken up. The problem is solved by treating the code that puts a thread to sleep and puts it on the waiting queue as a critical section that cannot be interrupted.

10. Why are deadlocks impossible if you prohibit incremental allocations of resources?

Deadlocks occur when one thread has locked one resource, another thread has locked a different resource, and each tries to lock the other's resource. If you prohibit incremental allocations, each thread requests all the resources it will lock at one time, and it either receives all its locks or receives none of them. In the former case, the thread cannot deadlock since it already has locked everything it will ever lock. In the latter case, the thread cannot deadlock since it holds no locks and thus cannot prevent any other thread from acquiring a lock.