Sample Final Exam Questions

CS 111

The three questions below are characteristic of the kinds of questions you will see on the CS 111 final exam. As you can see, the questions tend towards requiring you to apply knowledge, rather than repeat things said in the class or in the book. The questions are somewhat open ended, so the answers given here may not be the only ones that would be acceptable or lead to full credit on the final exam. But these answers represent what I would regard as a good, complete response.

The questions are in plain text, the answers in italics.

1. In the early 1990s, SUN Microsystems, the maker of the Solaris Operating System, wanted to move from the engineering desktop, where it was well established, to a broader market for personal productivity tools. The best personal productivity tools were all being written for Windows platforms, and SUN was on the wrong side of the applications/demand/volume cycle, which made getting those applications ported to Solaris a non-option.

One approach to their problem was to modify the version of Solaris that ran on x86 processors (the popular hardware platform for Windows) to be able to run Windows binaries without any alterations to those binaries. This would allow Sun to automatically offer all of the great applications that were available for Windows.

(a) What would have to be done to permit Windows binaries to be loaded into memory and executed on a Solaris/x86 system?

(b) What would have to be done to correctly execute the system calls that the Windows programs requested?

(c) How good might the performance of such a system be? Justify your answer.

(d) List another critical thing, besides supporting a new load module format and the basic system calls, that the system would have to be prepared to simulate? How might that be done?

(e) Could a similar approach work on a Solaris/PowerPC or Solaris/SPARC system? Why or why not?

The Solaris program loader and run-time loader would have to be modified to recognize the Windows load module and shared library formats, and to be able to load that code and data into a process' address space.

A new 2nd level trap handler would be written to intercept the Windows system calls, and pass it on to an emulation layer, which would try to simulate the effects of each Window's system call, using Solaris mechanisms.

The performance could be very good. User mode instructions would execute normally, and many of the system call simulations would probably be only a little more expensive than their native counterparts. A few operations would surely be expensive to simulate, but hopefully these would be rare.

We would also have to emulate the functionality of the Windows device drivers (specifically displays and printers). This could be extremely complex. It might be easier to provide our own DLLs to directly implement the higher level functionality on a Solaris display server. Other answers to this part could deal with the Windows registry or special Windows networking code. Other answers are also possible.

This approach depends on the fact that most (user mode) instructions are still directly executed by the CPU. If the CPU was a different instruction set architecture, this wouldn't work (a PowerPC or SPARC cannot execute x86 code). At this point we would be forced to go to machine emulation, which is much slower.

2. Waiting time in the ready queue is one metric for CPU scheduler performance. This question deals with finding metrics for the memory scheduler performance. In a landmark paper, Peter Denning asserted that the cost of any memory management strategy can be measured by the sum (over all processes) of the time-integral, for each process, of the number of page frames assigned to that process (or more simply the total number of page-seconds consumed by all of the processes in the system). If, for instance, a process has an image size of N pages and we leave it in memory for T seconds, the time-space cost of this decision is $N*T$ page-seconds.

(a) Justify this time-space product as a reasonable measure for the cost or performance of a memory management strategy.

(b) If we were to use pure demand-paging (without any pre-loading) rather than swapping, what (approximately) would be the time-space cost of allowing the same process to run for the same T seconds (if we assume that the we started with no pages and that the process faulted for its N pages uniformly over the T second period)?

(c) Is it likely that the demand paged process would require the same number of pages n as the swapped process N? Why or why not?

(d) The amount of time that a process is in memory is not merely the time it takes the process to run (T) but also includes the time it takes to swap the process in and out of memory. If we can swap N pages in or out in $T_s(N)$ seconds, the time-space product for swapping becomes $N*(T+2T_s(N))$. Suppose that we can fault-in or replace-out N pages in $T_p(N)$ seconds. Which is likely to take longer: swapping in N pages, or page-faulting for N pages? Why?

(e) What mathematical relationship must hold (between T, T_s, T_p, N and n) in order for demand paging to out-perform swapping? What does this mean about the programs, operating system, or hardware?

Beyond requiring you to understand what swapping and demand paging are, this question does not actually require you to remember much else about them. Rather, this is a test of your ability to do some simple mathematical reasoning about them
(a) If each a process uses fewer pages of memory, more processes can be fit in memory at one time, meaning that the scheduler is more likely to have ready tasks to run, and the CPU will be kept busy, and system throughput will be maximized. Thus, if the memory manager can run more processes in fewer pages, it is doing a good job.
*(b) At the beginning there would be no pages in memory, and by the end there would be N pages. If the fault rate is uniform, the area under that curve is T*N/2.*
(c) It is most likely that (n < N). Most programs exhibit reasonable code and data locality, so that during a small period of time they reference only a fraction of their pages (perhaps because they are executing one loop, calling only a few subroutines, and manipulating only a small amount of data).
(d) Tp(N) is likely to be greater than Ts(N) for a few reasons:

A process is usually swapped out to one contiguous region on disk, which means that the I/O involves little or no head motion. A process that is demand paging might have pages all over the disk.

There is start-up and completion over-head associated with each I/O operation. In swapping, there is likely to be only one large request, whereas in demand paging, there is likely to be one request per page, each with its own startup and completion overheads.

Beyond the I/O time, there is also an overhead associated with taking each page-fault.

(e) The cost of swapping is (from above) N(T+2Ts(N)). The corresponding cost of demand paging is*

n(T+2Tp(n))*
Demand paging will win if
n(T+2Tp(n)) < N*(T+2Ts(N))*
or (perhaps more clearly) if
n/N < (T+2Ts(N))/(T+2Tp(n))

This means that either n is much smaller than N (the programs exhibit good locality) or that Tp is not much greater than Ts (the performance penalty for page faulting and scattered I/O is low).

3. A simple scheduler for a single processor machine typically maintains a single queue of runnable processes to select from when it needs to find another process to run. It is possible to maintain multiple queues of runnable processes, instead.

(a) Why might it be desirable to have multiple queues of runnable processes?
(b) Give an example of how we might treat processes on different queues differently.
(c) Describe how the OS could determine which queue a particular process should be on.
(d) If there are multiple cores available to run processes, should there be:
(1). A separate queue for each core
(2). Multiple queues based on some other characteristic (such as that described in your answer to part (b)) shared by all cores

(3). Multiple queues per core

Why?

(a) The system might handle processes of very different types. They could have different dynamics, different priorities, or different scheduling requirements, such as real time scheduling. By placing processes of these different types on separate queues, the system can easily apply different scheduling algorithms to each queue, as appropriate for processes of the type kept in the queue.

(b) Processes on the real time queue might be ordered by their deadlines, to allow shortest job first scheduling; while processes on a more typical queue might be given round robin scheduling. Other answers are possible.

(c) In some cases, the programmer or the user who invokes a job needs to tell the system which queue to use for a process, either by the system calls used to create and run it, by environment variables, or through some other mechanism. Real time scheduling is one example. In other cases, such as where processes are assigned to different queues based on the typical time slice they require, the system can observe the process' dynamics, such as whether it usually uses up its entire time slice before blocking, and can move the process to the correct queue.

(d) If real time scheduling is being used, it is sensible to have one or more separate cores devoted to the real time scheduler, to ensure that deadlines are met. Otherwise, there is more liberty in the choice. Having particular queues tied to particular cores does have the advantage that processes in those queues will tend to be scheduled on the same core every time, which can have performance advantages if hardware caches on the cores survive long enough to allow re-scheduled processes to take advantage of data still stored in them. However, if the assignment of queues to cores is too rigid, some cores might be underutilized while others are swamped. An answer should demonstrate understanding of this issue and offer insight into how to make a decision. There are multiple possibilities here. One example: a meta-scheduler that examines current allocation of processes to cores and, based on the overall behavior of everything on a core, determines if it is over- or under-loaded. Another example: each process' behavior is examined, with processes that seem to show poor performance characteristics on their current cores (not getting a fair share of cycles, unusually long waiting times, whatever) being migrated to other cores' scheduling queues.