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. 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.

3. Many basic file systems (such as the Unix and DOS file systems we covered in class) tend to allow only a limited, pre-defined set of metadata attributes to be stored for each file. These include file ownership, size, access control, and a few other attributes. This allows the attributes to be stored in a fixed-size data structure, such as a DOS directory entry or a Unix inode. More advanced file systems sometimes support the concept of *extended attributes*. These are per-file metadata attributes that can be defined in the context of a single file. They may be defined at file creation time, or they may be added and removed from a file at any point during its lifetime. Often such file systems allow an arbitrary number and size of these extended attributes.
Assuming the file system is to be stored on a hard disk drive, discuss design strategies for storing these extended attributes on the disk. Outline at least three distinct ways in which one could handle the storage of these attributes on the disk drive. Consider issues of performance and efficient storage, at the minimum. Also discuss caching strategies for extended attributes. Will they be cached in exactly the same way as file data or in some other manner? Justify all of your design decisions.

Extended attributes could be stored in a metadata area, akin to the inode area used in Unix file systems. Alternately, they could be stored as if they were ordinary files, but with some attempt made to keep them close to the file they describe. A third alternative is to store them actually in the file they describe. There may be other interesting alternatives.

In terms of performance, the issue depends, to a large degree, on one's assumptions about how the extended attributes will be used. Will they be accessed more or less at the same time as the file data, or will they be accessed when one accesses basic attributes (such as when one is doing an extended listing of directory contents), or will they be accessed in some other pattern? It's hard to predict, but any reasonable argument for any choice can be acceptable. Assuming one predicts they will be accessed at the same time as file data, or close to it, then there is a strong need for locality with that file data. That choice would recommend keeping the extended attributes close to the file's data, perhaps in the same cylinder group. One could imagine parts of a cylinder group being devoted to extended attributes for the files in that cylinder group, but it's probably more efficient to treat all segments in the cylinder group as equally usable for files and extended attributes.

Given that one has no way of knowing, at design time, the size, content, or use pattern of extended attributes, it makes no sense to build a specialized cache for them, a la inode caching. The main question is whether to integrate the extended attribute cache into the basic block cache or not. Unless one has good reason to do otherwise, a single cache is likely to be more efficient in its use of RAM. However, if one expects lots of small extended attributes, much smaller than a block, it might be sensible to have a cache that works at a lower granularity. There are costs with doing so, of course.