# Stanford University Computer Science Department
# Fall 2002 Comprehensive Exam in Software Systems

Solutions by CS Ph.D. First Years (2006-10-19)

**Short Answers (3 each, 30 total)**

1.) With respect to transmission of a message from principal A to principal B, distinguish what is meant by authenticity, privacy, and integrity with respect to that message.

Authenticity – Was it really A who sent the message to B?
Privacy – Could a third party snoop and see the contents of the message or find out that the message was being sent?
Integrity – Did a third party alter the message while it was in transit?

2.) What is the point of multilevel paging?

To save a tremendous amount of space. Page tables are mostly empty; they are sparse data structures. If there were only one level, most slots would be blank.

3.) Give an example of a high-level language that uses explicit memory management, and state what facilities are provided for doing it.

C++. The **new** and **delete** operators are used to manage memory.

4.) What is an idempotent operation? Give an example of an idempotent operation, and contrast it with an example of a non-idempotent operation.

An idempotent operation is one whose results are identical no matter if it's executed once or many times. An example is 'x = x * 0'. A non-idempotent operation is 'x = x + 1'.

5.) Describe how failure recovery can be made easier by the use of idempotent operations.

If you experience a failure while you're in the middle of running the recovery code, there is no harm in running idempotent failure recovery operations again. This simplifies recovery code because it doesn't need to consider what happens if there is a failure during the recovery process itself.

6.) What's the difference between a preemptive and a non-preemptive process scheduler?

A preemptive scheduler forces processes to relinquish control of the CPU at certain times, while a non-preemptive scheduler doesn't; in a non-preemptive scheduler, the only way for some other process B to run while process A is running is if process A purposely yields control of the CPU.

7.) If a physical page is shared between two different processes, is it possible for the page to be read-write for one process and read-only for the other? If so, how, and if not, why not?

Yes, the page table entries for the two processes can point to the same physical page (usually mapped to different virtual addresses), and within each entry are permissions bits specific to the process. One process's entry for that page could be marked 'read-write' while another one's could be marked 'read-only'.

8.) Describe the difference between memory-mapped I/O and programmed I/O. Mention one typical application of each.

Memory-mapped I/O works by mapping the device into special memory addresses so that applications can interact with those devices by reading/writing those addresses. An example is a video card.

Programmed I/O requires special CPU instructions to read/write to a certain port and is not done through the memory system. Examples are legacy hard drives and mechanisms to write to a computer's CMOS memory.

9.) What is the difference between deadlock and starvation in a resource-allocation system?

Deadlock is when nobody gets a chance to run at all. Starvation is when one party never gets to run (or rarely gets to run) while other parties are running just fine. Starvation may be temporary and get relieved as time progresses (perhaps as load decreases), but deadlock is, by definition, permanent (until the system resets).

10.) A RAID system can fail if two or more of its drives crash within the same hour. Suppose that in a given hour, a drive can fail with probability *p*, and that drive failures are independent. What is the probability of a *k*-drive RAID system failing in a given hour?

P(*k*-drive RAID system failing in some hour) = P(2 or more drives crash within that hour)

P(2 or more drives crash) = 1 – P(exactly 1 drive crashes) – P(no drives crash)

P(exactly 1 drive crashes) = $k * p * (1 - p)^{k-1}$
P(no drives crash) = $(1 - p)^k$

Thus, P(failure) = P(2 or more drives crash) = $1 - k * p * (1 - p)^{k-1} - (1 - p)^k$

**Slightly Longer Answers (5 each, 30 total)**

11.) When a user program makes a (privileged) system call, in what important way(s) is the calling sequence different from that of calling another user-level procedure.

When calling a regular user-level procedure, the program saves caller-save registers onto the stack, pushes the return address and arguments onto the stack, and directly jumps into the code of the called procedure. However, when making a system call, the program sets up some register state and simply issues an interrupt to trap into the kernel. The kernel starts executing in a privileged state, saves the process's state so that it can be restored later, does whatever the system call requires, and returns back to the user program in a non-privileged state.

12.) Describe the *priority inversion* problem that arises in process scheduling or thread scheduling. Give a concrete example of circumstances that might cause priority inversion to occur.

Priority inversion is when a lower-priority process holds a lock (or some other resource) that a higher-priority process is waiting on, so the higher-priority process doesn't get to execute, thus inverting their priorities. A concrete example is the following: Suppose that there is a high priority process H and a low priority process L. L grabs a lock and then H tries to grab it. H can't run because L has the lock. However, now a medium priority process M appears, and because it has higher priority than L, it runs more often. L will eventually finish what it's doing and release the lock, but in the meantime, M runs much more often than H, even though it has lower priority than H, hence the priority inversion.

13.) Suppose you have a word-sized variable *Foo* that is shared among many concurrent threads. You would like to serialize accesses to this variable, to prevent conflicting updates. Unfortunately, you're writing in a language that doesn't provide language-level constructs for writing a monitor procedure (i.e., doesn't have features like Java's *synchronized* keyword).

(a) Assume your OS provides a mutex/locking facility. Using any suitable pseudocode syntax, write the pseudocode for *updateFoo(int newFoo)* using mutexes and/or locks.

```
int Foo;
mutex fooMutex;
updateFoo(int newFoo) {
  lock(&fooMutex);
  Foo = newFoo;
  unlock(&fooMutex);
}
```

(b) Assume your OS provides simple nonblocking atomic operations, like test-and-set or compare-and-swap, on word-sized operands. Write the pseudocode for *updateFoo(int*

*newFoo)* using nonblocking atomic operations. (Note! For part (b), *do not* use nonblocking operations to implement locks! Write true nonblocking code.)

```
int Foo;
updateFoo(int newFoo) {
  CAS(&Foo, Foo, newFoo); // compare-and-swap
}
```

14.) Suppose now that *Foo* is not an int but a data structure *FooStruct* that is several words long, and *updateFoo(FooStruct newFoo)* must atomically update the entire contents of this data structure. Can you still write only nonblocking code for *updateFoo*? If so, show the pseudocode. If not, explain why it can't be done. (Note: as in part (b) of the previous question, using atomic instructions to implement locks *does not* count as nonblocking code! Nonblocking code avoids spinwaits.)

No, because there is no way to atomically write more than 1 word. There might be interruptions from other threads while a *FooStruct* is in the process of being updated.

15.) Suppose that a CPU scheduling algorithm favors those processes that have used the least processor time in the recent past. Explain why this algorithm will favor I/O-bound processes; then explain why it will *not* permanently starve CPU-bound processes.

It will favor I/O-bound processes because those are likely to be blocking (waiting) on I/O lots of the time, and hence qualify for 'using the least processor time in the recent past.' However, it will not starve CPU-bound processes because after those I/O-bound processes get a chance to run for a bit, the CPU-bound processes now quality as 'using the least processor time in the recent past.' Also, because those processes are I/O-bound, they are likely to just do a little bit of work and then block for a long time, giving plenty of time for the CPU-bound processes to execute.

16.) Typically, a true LRU replacement policy can be used for managing blocks in the buffer cache of the file system, whereas only approximations of LRU are used to keep track of pages in a virtual memory system. Why is this so?

Speed. True LRU requires timestamps and is much more costly in terms of speed than an LRU approximation (such as the clock algorithm). When dealing with the buffer cache, the order of magnitude in speed is bounded by hard disk access times, which are much slower than CPU calculation times, so it's reasonable to implement true LRU. However, when dealing with virtual memory, which must be accessed more often and needs to be much faster, true LRU is simply too slow.

17.) Suppose a new CPU design has the following feature: there are multiple copies of the register file; at any given time, exactly one of these copies or "banks" can be accessed by user code; and a privileged instruction must be executed to change which "bank" is being used. How might the OS exploit such a CPU feature, and what benefit(s) would be gained?

The OS can exploit this feature to perform context switches more efficiently. Normally, to perform a context switch, the registers for the exiting process must be saved away in memory, usually somewhere in its own stack, and the register values for the entering process must be loaded from memory into the actual registers in the register file. This can be a slow operation. With multiple register files, the OS can simply execute a privileged instruction to switch banks when processes are being switched.