Stanford University Computer Science Department

Fall 2004 Comprehensive Exam in Software Systems

SOLUTIONS

1. **CLOSED BOOK**: no notes, textbook, computer, PDA, Internet access, etc.
2. **WRITE ONLY IN BLUE BOOKS: No credit** for answers written on these exam pages.
3. **EXPLAIN YOUR REASONING.** Answers with no explanation are insufficient.
4. The exam is designed to take about an hour if you budget 1 point per minute.
5. If you need to make assumptions to answer a question, *state them clearly*.

---

1) [5] Consider a virtual memory system with a single-level page table. The (dimensionless) page fault rate for a particular workload is *r;* the average DRAM access time is *d;* the average time to service a page fault is *f;* the (dimensionless) TLB hit rate is *h*. Write the expression for the average memory access time in this system.

This was tricky to compute, so I gave credit for either of two methods. If you break it down by whether a given access page faults or not, keeping in mind that a page fault takes time *f* plus two memory lookups (TLB miss + the actual access), you get $r(f + 2d) + (1-r)(hd + (1-h)2d)$. If you break it down by whether a given access hits in the TLB or not, you get $hd + (1-h)((1-r)2d + r(2d+f)$

2) [5] Consider three CPU scheduling disciplines: shortest-job-first, preemptive round-robin, and first-come-first-served. For each of these, if it is *starvation-free*, explain why; if it's not, *briefly* describe a scenario in which starvation could occur.

SJF is not starvation-free: short jobs could continue to arrive that always get favored over pending longer jobs. FCFS is starvation free as long as jobs terminate in bounded time. Preemptive RR is starvation-free because each process necessarily gets CPU time due to timer interrupts.

3) [4] Suppose a particular process makes a total of p page references, of which $n<=p$ are to distinct pages. (The ordering of page references is not known.) The process is allocated m frames of physical memory, initially all empty. In terms of these variables, give a *lower bound* and an *upper bound* on the number of page faults this process will experience, no matter what page-replacement strategy is used, and *explain your reasoning.*

No matter what, at least n page faults will occur since each page is touched at least once and the process's frames all start out empty. If $m>=n$, then after these n faults there will be no evictions, so the upper bound is n. If $m<n$, in the worst case every access after the m'th could still cause a fault (systematic eviction) so the worst case is p. It was OK to say the worst case is p as long as you qualified it with "if $m<n$".

4) [4] Debuggers like *gdb* let you set breakpoints that are triggered whenever a program variable, say *V,* is accessed or modified. How is this implemented? Under what circumstances, if any, does the usual implementation incur a performance cost when variables *other* than *V* are accessed? (To simplify the explanation, you may assume that we're only referring to global variables whose memory placement is known at load time.)

Page tables are modified to force a trap (usually due to an illegal page access) when the page containing *V* is accessed. Of course, since there may be other variables also stored on this page, a

check is required (comparing the page offset of the address) to confirm whether the access is to *V*; if not, it's a "false alarm". So a performance penalty is incurred on every access to the page containing *V*. Note that debuggers aren't compilers, and they don't require you to recompile your program to put in guard code. Saying that "exceptions are thrown" when the variable is accessed is just rephrasing the question, so it doesn't count. (*What* triggers the exception?)

5) [4] Give an example of a scenario where memory-mapped I/O makes more sense than programmed I/O, and *why* memory-mapped would be better. Then give an example of the opposite case.

A typical use of memory-mapped I/O is a video framebuffer since the framebuffer size is static and we're modifying entries in place. A typical use of PIO is short data transfers for character-mode devices or for accessing device registers that are used to synthesize a stream of data when sampled over time. Note that speed is not the main issue.

6) [4] To successfully prevent user programs from causing damage to other programs or the OS, hardware support is required. Name **two** hardware mechanisms in modern CPU's that supports this goal, and for each one, describe what specific kind(s) of damage it prevents.

Virtual memory, supported by the MMU, prevents processes from stomping on or reading each others' data. Privilege levels prevent user processes from executing certain instructions or code blocks that are reserved exclusively to the OS for inter-process resource management.

7) [4] Why would some OS's support multiple page sizes instead of just one page size? What additional page-management issues does this raise?

Using very large pages for OS code and shared read-only data reduces the number of page table entries and TLB entries that must be used to manage them. (Saying "reduces internal fragmentation" is not enough, since that's a basic principle of pages. The issue is *what situations* would benefit from this property.) However, it requires keeping track of whether pages "overlap", i.e. whether a given frame is part of a larger or a smaller page (some processors solve this by forbidding overlap, so that a single region can either be one big page or chopped up into small pages, but not both), and complicates physical address computation (the number of bits to use for page number and offset will depend on page length), page table walking and TLB lookup.

8) [4] Describe the mechanism of *priority inheritance* and give an example of the kind of problem it's intended to solve.

Consider threads 1, 2, 3 with decreasing priorities. 3 is running but has a resource needed by 1 (so 1 is blocked and 3 is about to release the resource so 1 can run). But 2 is on the ready queue, so 2 is scheduled and pre-empts 3. Now 1 cannot proceed because the resource it needs is still held by a lower-priority but not-running thread (3). **Note:** you need 3 threads to illustrate this.

9) [3] With respect to remote procedure calls (RPC), what is serialization (or marshalling) and why is it necessary? Are there cases where it is unnecessary?

Marshalling involves converting procedure call arguments (or return values) into a common representation for communication between machines that may have different architectures; for example, the packing of strings and the representation of floating point and integer values may differ between two machines, so marshalling converts them to a common intermediate form. It also follows pointers from data structures to collect all logical data structure members into a contiguous representation. If it is known that the representations of data structures and packing semantics are identical on both machines (which is a stronger requirement than just having the two machines use the same ISA), the first step can be omitted but the second is still needed.

10) [3] Describe one failure mode that might occur if a non-preemptive scheduler is used, and how it would be avoided with a preemptive scheduler.

One example: a program goes into an infinite loop and never does a blocking I/O or other operation that would yield the CPU. A preemptive scheduler would regain control on the next timer interrupt.

11) [3] A particular email message you're sending is so sensitive that you wish to both encrypt and sign it (both using public-key cryptography). Under what circumstances, if any, would you encrypt it first and then sign it? Under what circumstances, if any, would you sign it first and then encrypt it? (In other words, what's the practical effect of doing it one way vs. the other?)

Encrypt first, then sign, if it's OK for others to know who the sender is. Sign first, then encrypt, if you want only the designated receiver to be able to verify that you are the sender.

12) [3] True or false: all side-effect-free operations are idempotent, but not all idempotent operations are side-effect-free. Explain your answer concisely but completely (i.e. if true, explain why each part is true; if false, explain which part(s) are false and/or give counterexamples).

True. If an operation has no side effects, by definition executing it once is the same as executing it many times. On the other hand, an operation such as setting a variable to a specific value *does* have a side effect, yet is still idempotent.

13) [2] To avoid replay attacks, one can either use a randomly-generated nonce or a physical timestamp. Give one advantage of using a nonce over a timestamp, and one advantage of using a timestamp over a nonce. (You may assume that the resolution of physical timestamps is sufficient to avoid timestamp value collisions.)

Nonces better because synchronized clocks aren't required. Timestamps better because you don't have to remember every nonce you've ever seen.

14) [2] Describe one type of file access control that can be performed with access-control lists (such as AFS uses) but cannot be performed with file permissions (such as traditional Unix filesystems use).

One example: I can delegate (or revoke) write permission to a subset of users who are not all part of a common administrative group; you could allow everyone except one user to access a file.

15) [2] You're asked to take an existing Java program and rewrite it in C++. What benefit would you *gain* by doing so? What benefit, if any, would you *lose* by doing so?

Gain: C++ is generally a lot faster than Java since it's not interpreted. Lose: Java is type-safe, C++ isn't. Note, you lose *binary* portability but not necessarily *source* portability. If you claimed portability as your only lost property, you had to specify *binary* portability.

16) [2] What's the difference between a credential and a capability?

A credential proves that something (or someone) is what it says it is; a certificate signed by a trusted certificate issuer is an example. A capability allows the holder (who may be anyone) to take the particular action or use the particular resource specified by the capability, i.e. it is not specific to a principal.

17) [2] What's the difference between thrashing and deadlock?

Thrashing: processes can in theory make progress, but their aggregate resource needs exceed the ability of the scheduler; more time is spent shuffling resources than doing work, so processes make little or no forward progress. Deadlock: Processes cannot make progress even in principle since there is a loop in their logical waits-for resource graph.

18) [2] Give one example of how a filesystem might become corrupted, ***other than corruption of data in the files themselves***

The directory entry pointing to an inode or other metadata block may be lost. Or, the file metadata may not match the file's actual characteristics (e.g., length in bytes) because a failure occurred after

the file was updated but before the metadata was updated. Trashed "file descriptors" don't count, since they are not part of the filesystem and can usually be destroyed without resulting in filesystem inconsistency (though it may result in lost writes). Similarly, synchronization-related race conditions between clients writing a file do not damage the filesystem, they just leave it in a consistent-but-wrong (from program's point of view) state.

19) [2] Your C program has a bug that causes it to accidentally dereference a nonexistent array element, e.g. a[10] where array a[] has been statically declared as containing 8 elements. What happens when this bug occurs at runtime, and why?

You get a segmentation fault or illegal access fault, *from the OS.* C++ doesn't have runtime bounds checking so *you don't get an array bounds exception.*

THE END