Stanford University Computer Science Department

Fall 2003 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. **WRITE MAGIC NUMBER** on the cover of **EACH** blue book.
4. The exam is designed to take less than an hour. Most answers should be short.
5. **All questions are worth the same.**
6. If you need to make assumptions to answer a question, *state them clearly.*

---

1) 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.

2) 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.

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?

Serialization or marshalling refers to packing the call and its arguments into a portable format for transmission to the RPC server. It is necessary when you are not sure if the machine architecture of the server is the same as that of the client (and therefore whether types like integer or floating-point numbers have the same representation on both). Marshalling is not needed if the two machines have identical data representations and if the arguments in question do not contain pointers into the client's address space (which would not be safely dereferenceable on the server).

4) Describe a 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.

5) 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, each process usually has its own page table (or page table entries are annotated by PIDs), so the same virtual-to-physical mapping can have different access attributes in the different page tables.

6) 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.

Memory-mapped: reads and writes to a designated range of memory cause input or output to a device. A typical use is a video framebuffer since the framebuffer size is static and we're modifying entries in place. Programmed I/O: special instructions like "In" and "Out" cause data to be placed on special I/O lines that are decoded by devices. Uses include accessing device registers and short data transfers for character-mode devices since we are actually generating a data stream.

7) Least-frequently-used (LRU) replacement is often done for filesystem caches, but only a crude approximation of LRU is usually done for page tables. Why the difference?

It's too expensive in time and space to update the LRU information on every memory access (you could do this in the TLB, but you have to worry about flushing the info out on TLB evictions, which are still on much shorter timescales than filesystem caches). For filesystems, the cost of doing this update is usually small compared to the cost of doing the actual file operations.

8) 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.

9) *Priority inheritance* is when a lower-priority thread or process temporarily acquires the higher priority of another thread that is currently blocked. What problem does priority inheritance solve, and how does it solve it?

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. 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.
I/O bound processes frequently block till I/O completes, so when they become ready to run again, they will not have been running recently. CPU-bound processes have plenty of time to run during the I/O waits of the I/O bound processes.

10) You compile the following code in C using a typical Unix or Windows C compiler. Suppose you run it and call *func* with x set to –1 (negative one). What happens and why?

```
void func(int x)
{
    int a[10],b[10],c[10];
     ...code to initialize all elements of a[] to 1,b[] to 2,c[] to 3...
    printf("%d\n", b[x]);
}
```

Depending on the implementation of the compiler, this will print either "1" or (more likely) "3". No implementation will throw an exception or give a memory error, unless it stores automatic variables non-contiguously on the stack.

11) You translate the above code to Java, compile and run it, and again call *func*(–1). Does the behavior differ from the previous case, and if so, how? What is the reason for the difference in behavior (or lack thereof)?

The Java virtual machine specification requires it to do runtime bounds checking, so you'd get an array bounds exception being thrown.

12) Some OSs provided a system call RENAME to give a file a new name. Is there any difference between using this call to rename a file, vs. just copying the file to a new file with the new name and then deleting the old one? If so, describe the difference. If not, roughly sketch why there is no difference.

It's not the same. Rename modifies the file's metadata but does not move the file contents to a different set of disk blocks (unless the file is being moved across filesystems). (Rename is usually faster, but you had to explain *why* it's faster.)

13) With respect to a virtual memory system, give one argument in favor of large page sizes and one argument in favor of small page sizes. Give an example of an appropriate use of each type of page.

Small page sizes are useful when you want to avoid internal fragmentation (wasted space within a page, leading to wasted space overall). Most user data pages fit this description. Large pages are appropriate when you want to minimize page swapping because there is a large set of data that is infrequently swapped or is used all together; kernel image is a typical use for large pages.

14) Describe the structure of an inverted page table and how it differs from a conventional page table. When would you want to use an inverted page table?

Inverted page tables hash from virtual to physical address, allowing the page table to be the size of physical memory rather than the size of the virtual address space. If the virtual address space is very large and/or is sparsely used, inverted tables may be desirable.

15) When programming in a language that uses garbage collection such as Java, is it possible to have memory leaks? If not, why not? If so, give an example of a scenario that would cause a leak to occur.

Yes, circular data structures (such as self-referential linked lists) or references that go out of scope and are never used again cannot be garbage collected (because there is no way for the GC to prove that those data structures are no longer accessible).

THE END