SYS

Computer Science Department

Stanford University

Comprehensive Examination in Software Systems
Fall 2000

Read This First!

1. Write **all** answers in a blue book. **No credit** is given for answers written on these exam pages. *Don't panic!* This seems long, but most of the answers are very short.

2. Be sure to write your **MAGIC NUMBER** on the cover of **EACH** blue book you use.

This is an **OPEN BOOK** exam. You can't look up stuff on the Internet or ask other people, but you may use any books, notes, etc. you like, as well as a non-Internet-connected computer.

3. Use the point values of each question or subquestion to help plan your time. *Read the whole exam first; in case it's too long, do the ones you know right away.*

4. Justify your answers! Partial credit is given for good reasoning but a wrong answer, whereas it won't be given for a correct answer with incorrect or no reasoning. *State clearly* any additional assumptions you make.

1 Miscellaneous topics, concise answers [13]

a) [3] You compile the following code in C using a typical Unix or Windows C compiler. Suppose you run it and call the function *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]);
}
```

b) [3] Alice emails her stockbroker the following string in an email message: "Sell 100 shares of Oracle!" She encrypts and digitally signs the message so he knows it's authentic. Unknown to her, the evil Mitch is sniffing the email server. How can Mitch attack Alice? What could she have done to prevent it?

c) [3] In a remote procedure call (RPC) system, one possible problem is that it is impossible to distinguish between a really slow callee and a failed callee. Why might it be a bad idea to just retry the call after a certain amount of time has expired? (Ignore the possibility of overloading the network or RPC server.)

d) [4] A simple multithreaded program features *N* threads that share access to a common integer value *x*. To assure that only one thread at a time writes *x* (and that changes aren't lost), each thread's critical section looks like this:

```
        /* begin critical section */
        Lock(x);      /* will spin if necessary until lock is available */
        x = newIntegerValue;
        Unlock(x);
        /* end critical section */
```

Assume you have an instruction *CompareAndSwap2*, which is similar to the familiar atomic *CompareAndSwap*. CAS2 performs the following operation *atomically:*

```
        int CAS2(int *x, int *y, int oldXval, int newXval, int newYval)
        {
           if (*x == oldXval) {
              *x = newXval;
              *y = newYval;
              return SUCCESS;
           } else {
              /* don't change x or y */
              return FAILURE;
        }
```

Use CAS2 to rewrite the pseudocode for each thread's critical section *without using locks.* (Hint: use versioning.)

2 Memory Fragmentation [11]

Your superwizzy C libraries and runtime system provide standard system calls to allocate and free chunks of memory. There are no *a priori* restrictions on the size or alignment of memory that can be requested. The C function prototypes are roughly as follows:

```
typedef void *MemPtr;
MemPtr PtrAlloc(unsigned long sizeInBytes);
        /* PtrAlloc returns the NULL pointer if request can't be satisfied */
void PtrFree(MemPtr aPtr);
```

a) [3] Briefly describe the *memory fragmentation* problem and how it might cause a MemAlloc() request to fail even if there is enough unused memory to satisfy the request.

To alleviate this problem, you modify your runtime system and C libraries to support *handles*. A handle is a double-indirection to a block of memory:

```
typedef MemPtr *MemHandle;
        /* you can also think of it as: typedef void **MemHandle  */
MemHandle HndAlloc(unsigned long sizeInBytes);
void HandleFree(MemHandle aHandle);
```

b) [4] Explain how handles alleviate the fragmentation problem.

c) [4] Describe *two* performance impacts that arise when programmers routinely use handles. (**Hint:** one occurs frequently, the other relatively infrequently.)

3 Filesystems and Leases [12]

You're designing an NFS-like network file system for Unix that allows many clients to access and edit files on a network-connected remote fileserver. We'll refer to the server as *S* and three clients as *X,Y,Z*. When a client opens a file, the server sends a copy of the *whole* file to the client. In addition:

1. Any number of clients may simultaneously open a file for *read only* access.

2. If X opens the file for writing, a lock is set on the server so that future clients can *only* open that file for reading. When X closes the file, the lock is released so that future clients may open the file for writing. The entire act of writing X's changes and releasing the lock is atomic with respect to the server. The new contents of the file are *not* automatically sent to clients that have the file open for reading.

a) [3] Suppose X obtains a write lock on a file, and then crashes. When X reboots itself, it "forgets" which file(s) it had write locks on. What is the effect of this failure on the other clients and on the server?

To remedy the problem of (a), you suggest that the server use *leases*. A lease gives X the right to access the file *for a limited amount of time*. (As before, at most one client can hold a write lock on the file.) When that time expires, if X still wants to use the file, it must ask the server to renew the lease, otherwise the server will unilaterally terminate the lease (and release the write lock, if the leaseholder had one).

b) [4] Explain how leases fix the problem in the scenario of part (a), and explain any new effects seen by X in that scenario after it reboots.

c) [3] Suppose X is more careful: when it obtains a lease, it also records the fact that it is editing a particular file, and locally saves changes to that file as edits are in progress. X now crashes, reboots, and allows the user to recover the *local copy* of the file she was editing. Describe a scenario and the circumstances under which X might perceive a filesystem inconsistency.

d) [2] Assume that you can guarantee that any client's recovery time after a crash is at most *R*. Describe one possible way to avoid the inconsistency of part (c) , and describe its effect on the system.

4 Debugging Breakpoints [15]

You have been assigned the job of adding *data breakpoints* to an existing C debugger. The desired behavior is that the programmer can "mark" particular variables as being breakpoints: whenever a marked variable is *read or modified*, the debugger should take a breakpoint and allow the programmer to inspect the program's state, etc., then resume execution.

Assume that you are *not* allowed to make the programmer modify or recompile her source code, but you *do* have full access to the operating system and the runtime system, and in particular you can modify the virtual memory functions of the operating system (page fault handlers, page tables, etc.).

Also assume (as is the case in most implementations) that the compiler and linker arrange to store *global* variables in a designated memory pages that is known at link time, that all global variables will fit on one page, and that that page is not used for storing anything other than global variables.

a) [4] Suppose first that we only care about being able to mark global variables. Explain in detail how you would use the OS's virtual memory system to implement data breakpoints. In your explanation, keep in mind that only *some* global variables are likely to be marked.

b) [4] Qualitatively describe the impact on the overall speed of execution when the programmer marks a variable. What factor(s) dominate this impact? To what extent does the *number* of marked variables influence performance (assuming all marked variables are referenced equally often)?

c) [3] Suppose we instead want to support only *modify* breakpoints: a marked variable should cause a breakpoint only when its value is *modified*, not when it is read. What modifications could you make to your implementation to support modify-breakpoints more efficiently than read-breakpoints?

d) [4] Describe what additional complication(s) you would encounter if you also had to implement this feature for functions' local variables. Identify *at least one* important factor that would affect performance if this feature is added, above and beyond the performance effects already discussed.