Computer Science Department

Stanford University

Comprehensive Examination in Software Systems
Fall 2000

SOLUTIONS

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]);
}
```

Answer. 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.

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?

Answer. Mitch can replay the message later and cause another 100 shares to be sold; the broker can't distinguish Alice's message from Mitch's on the basis of content. Alice should have placed a timestamp or nonce (unique identifier) in her original message.

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.)

Answer. The function being performed by the RPC might not be idempotent (e.g. might have side effects). Without knowing if the first call succeeded, it's not necessarily safe to retry.

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.)

Answer.

```
/* let x be the variable to be atomically changed, v its version number */
do {
  tempV = v;
} until (CAS2(&v, &x, tempV, tempV+1, newIntegerValue) == SUCCESS);
```

Clarification. Note that we're not really exploiting CAS2 very well here (and for that matter, not exploiting locks very well in the original pseudocode); it's more useful when you want to atomically change a variable *from one known value to another* in the presence of possible race conditions. Because of this ambiguity, if you wrote correct code that *does* use locks, you still got partial credit; if your code simply would not give correct results (with or without locks), you got no credit.

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.

Answer. This problem arises when, through a sequence of allocations and deallocations, there is not enough *contiguous* memory to satisfy a request, even if there's enough total free memory.

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.

Answer. When a memory allocation request comes, the OS can shuffle around the blocks in the heap and modify the second-level indirect pointers accordingly. As long as applications always use double indirection to dereference memory, everything will work as long as the handles themselves don't change.

Note. To get full credit it was *necessary* to point out that handles imply the OS can move memory around without the program's knowledge. Some people had a serious misconception about how handles work, thinking of them as a table of free blocks rather than a double-indirection to a single contiguous block; this is *incorrect*, but if the answers to parts (b) and (c) were at least consistent, partial credit was usually given. Note that it clearly states in the question that "a handle is a double indirection to a block of memory", so there is really *no* justification for treating a handle as naming an array of noncontiguous blocks making up a single allocation request.

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

Answer. *Frequent:* every handle dereference is a double indirection, rather than a single indirection. This is significant since memory access is typically in the critical path of the performance of non-I/O-bound applications. *Infrequent:* when the memory allocator has to rearrange the heap and coalesce free blocks, a temporary stall (analogous to that introduced by garbage collection) could very likely occur. This problem is commonly observed in real systems that do implicit memory allocation. Partial credit was given for performance effects that, while they technically exist, are barely discernible compared to the above effects.

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?

Answer. The server cannot release the write lock on the file, so the file is locked against writing "forever".

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.

Answer. If X dies and reboots, and it forgets that it had a file locked for writing, after a while the file's lease expires. Since X doesn't know to renew it, after the lease expires the server revokes the lease and releases the write lock, and at that point others are free to write the file.

Any changes previously made by X are lost, however. (Some credit was lost for failing to mention this effect.)

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.

Answer. X locks file; X dies and restarts, but by the time it has restarted, the lease has expired and another client (say Y) has locked the file for writing. X now has local changes to the file that are inconsistent with what Y has written. If X tries to reacquire the write lock, it will lose its changes (since it will get the new copy of the file from the server). The answer "X still thinks it has the lease" is incorrect: if X really recorded the fact that it had a lease, it can determine after rebooting if the lease period has expired since it last renewed. Even so, no inconsistency would actually be seen by X until the above scenario occurs.

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.

Answer. If the current lease period is *L*, extend it to *L+R*. That is, if a lease renewal does not come, allow a "grace period" of length *R*, to account for the fact that X might have crashed and will renew the lease as soon as it recovers. The effect is that if X really doesn't care about the file (it didn't crash but doesn't care about renewing the lease), the other clients who are waiting for the write lock are forced to wait *R* longer than before. To get full credit you had to mention (at least in some general terms) the slowdown effect this causes on the system under normal operation.

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.

Answer. Mark the pages containing global variables as inaccessible by the user process. This will cause an OS trap (page fault) on every access to the page containing global variables. Use the faulting address (supplied by the page fault handler) to determine whether the faulting access was to a marked variable (very important); if so, transfer control to the debugger, otherwise resume the user program.

Some answers were written in a manner that assumes *every* memory access can be individually checked by a piece of code to determine if a breakpoint has been hit. This is impossible; the virtual memory system does not consist of arbitrary code that can be executed on every access. Marking pages invalid traps accesses and can *then* cause the inspection code to check if a breakpoint has been hit.

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)?

Answer. Every global variable reference, whether to a marked variable or not, will result in a page fault and a trip through the breakpoint logic. The dominant factor is the very high cost (in most systems) of taking a software interrupt: user-to-kernel crossing, context switch to new address space at higher privilege, etc., and the switch back to user space to resume the program. The number of variables does not matter (as long as they are contained on a single *page*) since any access to that page will fault. There may be a cost associated with determining *which* breakpoint was hit, and that cost *does* vary with the number of marked variables; but that cost is miniscule compared to the cost of handling the page fault. There is no associated disk access cost, since global-variable page faults will *not* generally cause disk access.

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?

Answer. Mark the appropriate pages read-only (rather than invalid/disallowed). Faults occur only when the page is modified, not accessed.

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.

Answer. Because local variables are allocated on the stack, their addresses are not known until the function declaring those variables is called. Detecting when this occurs would require some kind of support for setting code breakpoints at function entry points. The two deleterious effects on performance would be: (a) even if we can detect when a function is called and when it returns, we must now modify the page tables at each of those events. This is costly because it requires a change in privilege level or user-space-to-kernel-space crossing. (b) Assuming local variables tend to be referenced more frequently than global variables, which is usually the case, we would be trapping many more memory accesses compared with only marking globals.