Stanford University
Computer Science Department
CS 240 Quiz 1
Spring 2005

May 27, 2005

This is an open-book exam. You have 50 minutes to answer ten out of twelve questions. Write all of your answers directly on the paper. Make your answers as concise as possible. Sentence fragments ok.

NOTE: We will take off points if a correct answer also includes incorrect or irrelevant information. (I.e., don't put in everything you know in hopes of saying the correct buzzword.)

Question	Score
1 - 4	
5 - 8	
9 - 12	
total	

Stanford University Honor Code

In accordance with both the letter and the spirit of the Honor Code, I did not cheat on this exam nor will I assist someone else cheating.

Name and Stanford ID:

Signature:

Answer 10 of the following 12 questions and, in a sentence or two, *say why your answer holds.* (5 points each).

1. Eraser ignores loads and stores off of the stack pointer. Explain what problem this causes and how you would have to modify Eraser to fix this shortcoming.

   Problem: if a thread passes an address of a stack variable to another thread, Eraser will ignore the loads and stores of the first thread, which will miss potential races. Eraser will have to track these and, in addition, it will have to "reset" the stack space used by a procedure call when it returns, for the same reason that it resets freed memory.

2. In Ousterhout's talk, on the "Why threads are hard, cont'd" slide, he has two pictures about "deadlock!" Using your understanding of the Mesa paper, what causes the deadlock in these pictures? Could you usually statically detect these cases?

   Both pictures were discussed in the Mesa paper:

   (a) Thread 1 calls into monitor A, then waits on a condition variable in monitor B, which can unfortunately only be signaled by thread 2 who must reach B by first going through monitor A (which it can't).

   (b) This is straight circularity: T1 is in A and tries to enter B, T2 is in B and tries to enter A.

   In the absence of function pointers you can detect that these deadlocks can possibly happen by just looking at the callgraph.

3. Assume we have the trivial MESA monitor:

```
---------------------------- Monitor M ------------------------
entry foo()
end;
---------------------------------------------------------------
```

   which is called by the following code:

```
1:         main() {
2:             p1 = fork foo();
3:             p2 = fork foo();
4:             join p1
5:             join p2
       }
```

   If all threads run on the Mesa system with the same priority, how will the scheduler interleave the threads as the code runs? MAKE SURE YOU SAY WHY.

   Recall that:

   (a) Mesa does round-robin, FIFO scheduling of threads that have the same priority.

(b) *Mesa only switches out a thread when either (1) a higher priority one becomes runnable (which doesn't happen here) or (2) when the running thread blocks.*

Thus, the schedule is:

(a) *The "first" thread runs lines 1,2,3 and blocks on the join at line 4.*

(b) *p1 will run and complete (it is the next one in the FIFO run queue).*

(c) *p2 will run and complete.*

(d) *The first thread will resume and complete.*

4. Assume a cooperative thread system, fill in the code for a lock function (note: it must be < 5 lines!):

```
void lock(int *l) {
        while(*l == 0)
                yield();
        *l = 1;
}
```

What is the main reason that a capriccio program might behave correctly when it forgets to acquire a lock, but incorrectly when you add a lock call?

Lock can yield, which will allow another thread to run, which might violate invariants that are not protected by the lock. Without the lock then (assuming no other yield) no other thread will be able to run, preserving the invariants.

5. The VMS guys read the ESX paper and get excited about using taxation to adjust each process's RSS. Instead of computing f like Carl does they plan to either: (1) use the rate at which pages get evicted from a process's RSS as f or (2) use the rate at which pages get rescued back to a process's RSS as a measure of f. Why would you prefer (2) over (1)?

A couple of positive things about (2) over (1):

(a) *Since we rescued pages, then we were going to reuse them. This means we will benefit the current process for sure. In constrast, eviction may be kicking out things we will not reuse. Thus, growing may not help.*

(b) *Worse, basing f on eviction gives no performance isolation. An extreme example: our machine has 100MB but process A is cyclically iterating through a 1GB array. Growing A's RSS based on eviction, will blow out all other processes' caches, and also give no benefit to A.*

(c) *Basing on rescue also has the benefit that we may be saved from subsequent 200usec rescues costs.*

6. Assume you run a process that has a 513K text segment on the Alpha system described in the superpage paper. List the different actions in the superpage system that could occur when you jump to the first instruction in this process, ordered from best case to worst.

   There were various arguable orderings. We took a bunch as long as it was plausible. A crude order:

   (a) *The process will get a 512K reservation (note: it will not get anything larger!).*

   (b) *A 4MB reservation is preempted and broken down to 512K.*

   (c) *The page deamon is run and 512K is recovered.*

   (d) *Not enough space is available so we can only do 64K or (worse) there is no reservation.*

   (e) *Allocate 8K page.*

7. ESX: Assume we did an experiment similar to Figure 8. Give a situation that would allow: (1) our alloc and active lines to increase and mirror each other but the balloon line to stay the same, (2) the active line to increase dramatically, but alloc remain the same, and (3) balloon remains the same but alloc decreases dramatically.

   (a) *There is plenty of memory, and other processes are idle. The more we actively use, the more we get.*

   (b) *One process is running and is already using all the memory on the system; more active doesn't matter if there is no more bits. Another possibility: this process is using all it was configured for. Another: memory is low, and everyone is even busier.*

   (c) *Same as 1.*

8. What would be the difficulty in applying nooks to (1) the Linux's process scheduler and (2) qsort?

   (a) *Nooks as described must kill any component that is screwed up, clear its state, and then restart it. Resetting the scheduler will wipe out all processes on the system, which is the moral equivalent of rebooting which nooks was designed to prevent.*

   (b) *qsort's bug is probably deterministic; so when you restart it, you will hit hte same error.*

9. For the following code:

```
1: int *p;
2:
3: p = malloc(10);
4: p += 11;
5:  *p = 1;
```

Explain how the Lam cred compiler will rewrite line 4 and what will happen when this code runs. Explain how the Rinard alteration of the cred compiler will rewrite line 5 and what will happen when the code runs.

(a) *Lam will do a lookup of p to see if its inbounds, check to see if the increment will take it out of bounds and, if so, generate an OOB object (which it will in this case).*

(b) *The rinard compiler will check to see if the object is out of bounds (it is) and, if so, discard the write.*

10. VTRR: Figures 2 and 3 use averaging in a dangerous way; how could this make VTRR look much better than it is?

    They take the average of the highest service time and negative service time, which will potentially damp out any spectacularly bad errors if they are usually not so bad.

11. Let's say you have a stock BSD system (w/o Mogul's modifications) and you know it is doing packet forwarding. How could you hack ESX to prevent this forwarding system from livelocking *without* reducing MLFRR much?

    For the forwarding system you know that one packet in should = one packet out. Thus, livelock = many packets that come in do not make it out (occassionally you may lose things even not under livelock). So the key is to figure out roughly what the MLFRR is and then never exceed that packet input rate. A simple approach is to gradually increase the packet delivery rate until you see a non-negligible loss rate. A more clever approach would be to do something like a binary search.

12. The livelock paper states you can use one of two approaches in solving livelock: "(1) do (almost) everything at high IPL, or do (almost) nothing at high IPL." As they describe these approaches, in what sense are they actually the same?

    Both will shut off work generation when processing packets. The first by setting a flag and disabling interrupts, the second by doing everything in the interrupt handler, also preventing further interrupts from occuring until it is finished.