Stanford University
Computer Science Department
CS 240 Quiz 1
Spring 2006

May 12, 2006

This is an open-book exam. You have 50 minutes to answer eight out of ten 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 - 5	
6 - 10	
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 eight of the following ten questions and, in a sentence or two, *say why your answer holds.* (5 points each).

1. Given a program P that runs on Caprpriccio and does no explicit locking of its own. What does P correspond to in terms of MESA? What is the most straightforward way you could adapt Eraser to find race conditions in such programs? (Hint: think of how it checked OS code.) How useful do you expect this to be?

   P = one big monitor with one implicit condition variable, yield = wait statements on this variable. Since there are no locks, Eraser wouldn't be able to check much. You could have one lock for the "monitor" that was implicitly acquired when a thread starts running and released at yield point. However, this isn't very useful since it would be held whenever P ran. Finding races in this case would actually require looking for violations of invariants, which Eraser is not setup to do.
   Grading: *(+2) saying that the program was one big monitor with words implying that this gave a form of mutual exclusion through non-preemption or that whatever thread was running had exclusive access, etc., (+1) by tying the wait and notify to thread yielding and scheduling and (+2) for identifying that Eraser wouldn't be useful because there are no locks (except maybe the one big monitor lock shared by everyone) in the program, so hence the lockset algorithm is useless.*

2. Consider the Therac threading model. How does it compare to Events in Ousterhout? To threads in Capriccio? Does it contradict or support the arguments of these people?

   Therac is all screwed up, so we had to take a variety of answers. In some sense you can see it being event-driven, since there is a sort-of event loop that decides what thread to run. In another sense it is standard pre-emptive thread with the standard default (nothing is a critical section). It can be seen as a vindication of the pre-emptive threads are really hard mantra. But programmer would have probably screwed up any model, so things are a bit unclear...
   Grading *roughly: (+2) to make the connection between events and how Therac dispatches certain tasks, (+2) observing that the threads in Cappricio are non-preemptive (not preemptive!) and has nothing to do with the preemption problems in Therac (although talking about implicit yields was ok), (+1) making a lucid argument why this does or doesn't support Ousterhout's claim; the main issue here was confusing Cappricio's threads with pre-emptive threads, since the claim of the authors of the Cappricio paper is that you get the benefits of events (we avoid the problems of pre-emption) without the fuss of continuations.*

3. Consider very-broken code that uses the "double-check" lock idiom:

```
0:   int *p = 0;
```

```
....
1:   if(!p)
2:      lock(l);
3:      if(!p) {
4:         int *t = malloc(sizeof *t);
5:         *t = 3;
6:         p = t;
7:      }
8:      unlock(l);
9:   }
10:  x = x / *p;
```

How can an optimizing compiler of the sort that Boehm complains about interact badly with the use of the `t` temporary? Assume you don't use an optimizing compiler: if another thread has done a `free` previously, what can happen if the initializing thread gets context switched immediately after line 6 and another thread executes line 10 on another processor?

It could remove the temporary assignment and assign directly to `p`*, which means that another thread could see that* `p` *was not null before its contents were initialized. If the free had happened on another processor, then that processor could still have a stale copy of that memory in its cache. If this memory was then reused by the malloc and assigned to* `p` *and the second thread ran on the processor with the stale data it could see the out-of-date copy. This would be hard to debug. (Note, in reality that malloc and free probably use locks which will probably cause memory synchronization, which would prevent this bug from showing up.*
Grading: *roughly: (+2.5, rounded up with the total sum) talking about how an optimizer could screw things up; main example was optimizing away* `t`*, but other another example included reording the statements in the critical section (a problem since the access to* `p` *on line 10 is not protected by a lock). Some people said that the compiler could move the call to* `malloc` *outside of the critical section; for a general function this is completely false; because of potential side-effects the compiler cannot reorder function calls unless it can perform a whole-program analysis and determine that functions can be reordered because one does not effect the other. Because* `malloc` *is often an intrinsic function, the compiler can place the call to* `malloc` *outside the critical section, but the assignment to* `p` *must be done within the critical section (we don't believe, however, that a compiler like* `gcc` *does this in practice). Another possibility is that the inner* `if` *statement could be optimized away, causing issues. This is potentially possible since* `p` *is not declared* `volatile`*. A variety of other answers were closely evaluated... (+2.5, rounded up with the total sum) the main issue with the call to* `free` *is that the pointer value can be stale across processors (for many reasons). Other answers were also considered on a case by case basis. Partial credit was given (removed) for technical points that were unclear or incorrect.)*

4. Your 240 cohort sees the number 98.2 in the VMS paper and says "that is strange." Why? Assume the methodology that obtained it was accurate: What is a plausible

reason the number is what it is?

This is the average number of pages written. They claim the high-low limit is around 100. One plausible explanation is that pages with pending I/O operations were written and then rewritten when the I/O ops completed.
***Grading** roughly: (+2) that it was clear that 98.2 was the number of pages written. Some answers would conflate this number with the number of pages read, or simply did not clearly indicate they understood what this number was. (+1) about some discussion of the modifies list, or something to the effect of where the pages came from and that they were batched, or a reasonable explanation of why the number was around 100 (+2) a flexible answer to identify why the number is 98.2 as opposed to something like 100 (reasoning in an experimental way about why the number was the way it was).*

5. Superpages: Jimbob says the most important thing to know about the future is when an object will die. Bobjim says that the most important thing is how an object will be populated. How could you use these facts? Give an experimental reason for why one of these guys is more right.

   You could use death times for placement: put things that will die together near each other so that death frees up a lot of space. You could use population information to decide on whether to do eager promotion or not. From the experiments, the overhead of lazy promotion and general management doesn't seem to be such a big deal, so probably placement is more important (since it lets you use superpages when memory contention is much higher and the benefit of superpages for many apps is non-trivial.).
   ***Grading**: (+2) to reason about how Jimbob's policy can be used to effectively reclaim pages, etc., especially in the context of how this helps management of superpages. Many answers did not talk about how this was useful from a page management perspective (this question is about superpages!). (+2) to reason about Bobjim's policy, and how it could be used for eager promotion, or something useful for page management (again many answers ignored the effect of the policy on superpages). (+1) a reasonable argument, one way or another, of why one policy was better than the other by comparing the benefits of both as put in the previous two points. Some answers simply said that one was better than the other without mentioning what the benefits of one or both facts were. Both facts are potentially useful, and to get full credit you needed to discuss both of them.*

6. Most OSes don't have superpages. VMware hires Navarro to hack ESX so that ESX will trick guest OSes into using superpages. At a high-level: how would he do so? Do you expect this approach to work well? Why or why not?

   What he would want to do is to map the PPN's of contiguous VPNs to contiguous

MPNs and insert this into shadow page tables to they get put in the TLB at miss time. However, this probably won't be that effective: the guest OS will just pick PPNs it wants, which will almost certainly not map to contiguous MPNs. He would have to do some sort of compaction scheme to see any real continuity.

7. What would suck if ESX only used a "slow" average for measuring active memory?

   It could not adapt to sharp increases in memory usage, which means that these programs would be starved for mem and run much slower than necessary.

8. You use the Safe-C compiler in the Rinard et al papers to implement memory protection in Nooks. Give three things you are pretty sure would change or not change in Figure 8, "Nooks" (and why!).

   Page tracking and object tracking will probably not change that much since you still have to do this. The time it takes to run the code will likely go way down since you won't need to use different address spaces to enforce protection. On the other hand, the actual code will run a lot slower so its hard to say if things will be faster or slower.

9. "Take from the head, put on the tail." Give three places we have seen such a data structure and, in each case, what high-level policy it was attempting to achieve. Give one place where we might have seen this but did not (and why).

   This is how you implement standard LRU. We saw this in VMS for free and modify list. In superpages for the reservation lists. We did not see this for how ESX revoked pages — it used random to get around the double-paging problem.

10. You see the code:

```
1:    p = p + k;
2:    if(p == q) {
3:          x = *p;
4:          y = *q;
5:    }
```

    How could the dereference on line 3 be illegal, but the one on line 4 be legal? Give pseudo-code for how the expression on line 1 will be rewritten by the failure-oblivious compiler.

    Error: `p` *points to a valid object O1, but the addition at line 1 pushes it so far it*

now points into the object O2 pointed to by `q`*. The raw addresses these pointers contain will be the same but the logical objects the programmer intended them to point to are different.*

```
if(p is an oob)
        oob.offset = k;
else {
        // lookup object that p points to and get
        // its base and size
        b = hesh(p).base;
        u = b + hash(p).nbytes;
        // p is inbounds: do as normal.
        if(p >= b && p < u)
                p = p + k;
        // p is out of bounds, make an OOB object and
        // record where things are.
        else
                oob = new oob;
                oob.base = b;
                oob.u    = u;
                oob.offset = p + k - b;
                p = oob;
```