Stanford University
Computer Science Department
CS 240 Quiz 1
Spring 2004

May 6, 2004

This is an open-book exam. You have 50 minutes to answer eight out of nine 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 - 2	
3 - 4	
5 - 6	
7 - 8	
9 - 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 8 of the following 10 questions and, in a sentence or two, *say why your answer holds.* (5 points each).

1. Eraser uses local locksets — i.e., the locks in a threads lockset are only those that it acquired. Of course, the set of locks that are currently locked is a global property that does not depend on which thread is running. What would happen if Eraser used this global rather than local lockset when checking for races?

   The global lockset will always be strictly larger than the local one so you will get false negatives, depending on the scheduling. For example, the following contrived execution sequence would prevent eraser from flagging that T2 does an unprotected access to variable x:

```
T1                            T2
lock(a);
         --->  switch -->
                             x++;
         <---  switch <---
x++;
unlock(a);
```

2. Linux has a "big kernel lock" (the BKL) originally used to more-or-less turn the OS into one large monitor. Using your knowledge of Mesa: what must Linux do (and why) if a kernel thread holds the BKL and sleeps (i.e., puts itself on a block queue and allows another thread to run)? Give an example of a problem this can cause. (Note: you only need to understand Mesa to answer this question, the details of Linux are completely irrelevant.)

   If a monitor sleeps holding the lock waiting on a condition it must release it otherwise no other thread can enter the monitor to make the condition true. This is also true in linux with the addition that there is no other monitor, so it must always release the lock since otherwise the OS cannot run at all. So at each sleep point, the kernel releases the BLK and then when the thread wakes up, reaquires it. The problem this causes is that this breaks the critical section, which can cause a race condition.

3. What would the dual of a monitored record be in an MPP system? What is the dual of a message ID in the procedural system?

   Monitored record: you would need a thread+event loop for each record. This would be a very large overhead.

   Message id: tells you which specific message send you are replying to, similar to the return address in a procedural system which tells you which specific callsite for "foo" you are returning to (strictly speaking the return address + stack pointer).

4. The Capriccio paper states that "larger path lengths require fewer checkpoints but more stack linking." What is the intuition behind this statement?

   The first part is easy: the larger the limit, the more calls you can can do before checking that you've exceeded the stack. The right way to look at the second part is that as you increase the depth of the call chain between checks, the chance that there is a worst case path that will exceed your current stack increases, forcing you to allocate another stack.

5. Assume we only have two jobs, a and b. The VTRR paper would claim that if $W_a(t)/S_a = W_b(t)/S_b$ then we have "perfect fairness." Prove this is true.

   Wa(t)/Sa = Wb(t)/Sb implies:

   (a) *Sb Wa = Sa Wb.*
   (b) *t = Wa(t) + Wb(t).*
   (c) *Sum(Si) = Sa + Sb*

   We need to show that given these that the error is zero. I.e.:

```
        0 = Wa(t) - t * Sa / Sum(Si)
          = Wa   - (Wa+Wb)*Sa/(Sa+Sb)
          = (Sa+Sb)Wa - (Sa Wa + Sa Wb)
```

```
= Sa Wa + Sb Wa - (Sa Wa - Sa Wb)
= Sb Wa - Sa Wb
= Sa Wb - Sa Wb
= 0.
```

Many students proved the opposite direction (that above equality holds true if there is perfect fairness. If that proof is fully correct, we generously gave 4 out of 5 points since the proof is almost if-and-only-if.

Many students merely restated the problem in their own words. We awarded one point for that.

6. VMS has several different parameters. If you were to try to hurt system performance the most by setting a single parameter to its worst value, which one and value would it be? Explain your reasoning in terms of numbers given in the paper. You will need to make assumptions: make sure they are reasonable, and make sure to justify them.

   This question is very open-ended. Setting RSS to 0 will probably have the worst overhead. First, since the system will likely break immediately. Second, even if it does not break, every single memory reference will require a list removal and insertion. This would happen every few instructions (if not every instruction given that instructions reside in memory). Assuming 10ms disk access time, then the paper states that a rescue takes 200usec, which means that if we do it more than 50 times we pay more than for a single disk access. The alternative would be setting RSS to the size of memory, degenerating to FIFO, but from the graphs FIFO will most likely not cause us to take a page fault every 50 memory references so would still be better.

   We took several other answers, as long as you justified them. Lots of people said setting the modified list to 0 would cause a lot of problems because of writes, but while this would hurt performance, writes are asynchronous (unlike reads) so would almost certainly be not as serious. Usually this got two points, unless there was a good discussion of assumptions.

7. Describe the worst-case address space layout that maximizes VMS page table overhead (i.e., the amount of memory consumed by page tables given the amount of memory mapped). Be very specific.

*VMS uses linear page tables with 32-bit entries. Pages are 512 bytes long, which means they can hold 128 PTE entries. So a bad address space layout would be to allocate every 128th page forcing it to use an entire page to hold a single PTE. An even worse one would be to maximize the overhead of the system page table as well, which needs one entry to map a page of a P0 or P1 page table. We can similarly maximize overhead by having it map "empty" PTEs by allocating pages 128*128th page. This is really bad since the system page table resides in physical memory and so cannot be page.*

8. Assume superpage management (promotion, demotion, etc) is free. Describe a workload that will perform much worse on the Navarro superpage system as compared to FreeBSD.

   The most straightforward: Navarro puts closed files on the inactive list where they can be moved to the cache list under continuity pressure. A simple workload would be (1) a program that runs repeatedly and accesses the same files running concurrently with (2) an application that needs large superpages (e.g., FFTW) and hence steals these pages for its own ends.

9. Figure 8(c) of the ESX paper (the graph for the citrix server): explain what the synchronized spike in the "alloc," "active" and valley in "balloon" at (roughly) the 39-42 minute mark imply about what is happening in VMware and in the Citrix server. Assuming the system had 2GB rather than 1GB: what would the shape of the lines be?

   alloc spikes, which means the application is using more memory. The active spike tracks that of alloc closely, which means it is a good estimation of alloc. And the balloon goes down inversely, which means that (1) this application is getting memory (most likely because it has a lower idle count than other ones) and (2) it had a reasonable amont of memory in its balloon and (3) the memory it is using is coming by unballooning. If the system had 2GB then the active line would be *the same, the alloc line would be flat (all applications would have their max) and the balloon line would be flat as a result.*

10. Most OSes have some sort of `mpin(void *va, unsigned len)` system call, which will pin a range of addresses in physical memory. Explain how to use this system call to replace the need to write a balloon driver. (NOTE: make sure to describe how to communicate with ESX.) What is the advantage/disadvantage of this approach?

    You have to open a secret channel with ESX. The most straightforward is probably just open a socket connnection. The application would malloc a big region and pin it. It would then unpin it to shrink. Downsides: (1) the application must run before it can pin/unpin memory, which will be much less under ESX's control than a device driver (2) pages still might not be reused quickly by OS when unpinned if they are considered "accessed" (3) the application will have to signal to ESX which pages it pinned — possibly by writing special values into the pages (a little shady), finding a way to walk the page tables yourself to do the virtual to physical translation, or augmenting ESX to do it by inspecting the guest's page table when it received communications over its secret channel.