Stanford University
Computer Science Department
CS 240 Quiz 1 Spring 2009

June 1, 2009

!!!!!! SKIP 20 POINTS WORTH OF QUESTIONS. !!!!!

This is an open-book exam. You have 75 minutes. Cross out the questions you skip. 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 (25 points)	
6-10 (25 points)	
11-13 (40 points)	
total (max: 70 points):	

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:

Short answer questions: in a sentence or two, *say why your answer holds.* (5 points each).

1. You change Eraser: when it detects an access to memory location m will cause the lockset l associated with m to become empty, it instead acquires the locks in l and releases them after the access. How effective is this at removing errors?

   Not very. If it literally acquires and releases before each load and store, it provides no protection. It might provide some protection if it acquired/released around each locgical store since writing 8-bits would require a read-modify-write cycle. Also, having Eraser acquire locks in a random order can easily lead to deadlocks.

2. Your ex-140 partner rewrites Eraser so it tracks a lockset for each *bit* of memory. Roughly, what will happen to the memory overhead? For the machine used in the Eraser paper, can doing things this way actually cause Eraser to miss errors?

   32x memory increase in shadow memory. Possibly increase in other lockset related data structures, but the overhead from shadow memory should dominate. It can miss errors now: assume in word w the program consistently acquires lock $l1$ for bit 1 and $l0$ for bit 0. Eraser will flag no error. However, since the program will write words (or bytes) if two concurrent writes happen, one intending to update 1 the other 0 then the last writer will win, and one of the updates will be lost.

   On can also argue that this will not cause Eraser to miss errors if you explain correctly that since there are only 32-bit loads and stores, the locksets for all individual bits in a word will always have identical values, so there would be no difference to the original approach.

3. Your MESA code has two monitors, M1:

   ```
   entry foo1() { bar1(); }
   entry foo2() { bar2(); }
   ```

   And M2:

   ```
   condition c;
   entry bar1() { wait(c); }
   entry bar2() { signal(c); } }
   ```

   What can happen? Why does MESA work this way compared to the alternative?

   Deadlock:

   ```
   Thread 1        Thread 2

   M1.foo1();
                     M1.foo2();
   ```

M1.foo1() will sleep in M2.bar1(), but M1's monitor lock will not be released. If the only other call into M2 is through M1.foo2(), then we can never run signal. If M1's lock was released on a wait in another monitor then you would have to guarantee the monitor invariant before any external call: (1) this is hard (bugs) and (2) if `foo` *doesn't block today, what about tommorrow?*

4. What two fixes should you make to the allocation code given in the MESA paper so that it will work with their system?

   Mistake in the problem: one fix. You should broadcast when you get more space: a given free can satisfy more than one alloc. (See Section 4.1 of the paper)

5. Assume that on your machine a read always returns the value of the last write. If you compile and run the code below using a compiler similar to the one used in the Boehm paper:

```
          % initial: a = 0, b = 1
        thread 1  |    thread 2
   ---------------|---------------------
     r1 = a;      |   r3 = b;
     r2 = a;      |   a = r3;
    if(r1 == r2)  |
          b = 2;  |
   ---------------|------------------
```

   Give the smallest and largest number of registers `r1`, `r2`, and `r3` that could hold the value "2" after this code runs (and explain how).

   Zero registers hold 2 if thread 2 runs to completion first. All three can hold if the compiler optimizes. Since there are no lock/unlock calls it's perfectly legal for the compiler to redo thread 1's code to remove the if-statement (which is tautological for sequential code) and hoist the assignment:

```
b = 2;
r1 = a;
r2 = a;
```

   Then the following interleaving gives all 2's:

```
b = 2;
               ------->
                              r3 = b;
                              a = r3
               <-------
r1 = a;
r2 = a;
```

   And there you have it.

6. Assume you check this code using Eraser:

```
 Thread1          |          Thread 2
my_lock(l);       |         my_lock(l);
x++;              |         x++;
my_unlock(l);     |         my_unlock(l);
y = x + 1;        |
```

   Assume further that the code actually executes and Eraser flags no error — what plausible thing could be happening? (Hint: think like the Boehm paper and make sure your answer causes Eraser to miss errors rather than flag them.)

   If the compiler (1) does not know your lock functions actually locks things and (2) determines they do not modify `x` *or* `y` *it could rewrite*

```
my_lock(l);
x++;
my_unlock(l);
y = x + 1;
```

   to be:

```
my_lock(l);
x++;
y = x + 1;
my_unlock(l);
```

   making it protected by `l` *(by happenstance).*

   We gave credit for some other plausible compiler tricks as well.

7. An excellent question came up after class: for Superpages, Table 3: `galgel`, `C4` and `SP` have 0.00 as their entry in column "`4MB`." What does this mean? What is the likely explanation? (Note, this might be a little tricky: make sure you know what the number means.)

   0.00 means there was no speedup when 4MB superpages were used for everything. In Table 1 you can see that for these benchmarks (almost) never any 4MB superpage gets used. One possible reason: recall superpages must have all the same protections and only get promoted when they are fully populated. If one page in 4MB has a different reference bit, dirty bit, or is not paged in, then the whole thing won't get promoted to 4MB superpage. Also if code and stack/data are next to each other you probably can't promote since they will have different protections.

8. ESX: Figure 6 (the toucher thread experiment): let's say you eliminate both of the fast averages: where in the figure would this be worst for the toucher thread? Let's say you eliminate the slow averages, where would this be worse for the toucher thread?

Worst for no fast: the sharp uptick at 9min. It won't be able to adjust up to that so will thrash. Worst for no slow: the valley around 20: it would immediately drop the usage down and getting it back up might be an issue. (proper explanation was important here)

9. You change the failure oblivious code to return a random value on out-of-bound reads:

   ```
   T failure_oblivious_read_read(T *p) {
     if(inbounds(p))
         return *p;
     else
         return random();
   ```

   Explain what will (likely) happen for the midnight commander bug as compared to the approach they use.

   Midnight commander `strcat`*'d into an uninitialized buffer. Strcat will walk the string looking for the end (a 0 byte). Using their values it will hit is pretty soon after going off the end of the string. If you return random, it will loop a long time before random returns 0.*

10. Give a four line or less C code snippet that contains an error that will cause problems without failure oblivious, but will work "acceptably" with it. (Explain why!)

    Full marks required having a piece of computation where discarding results would allow it to still produce acceptable results. You had to give an intuition for why these results were acceptable (e.g., just saying it would truncate output was not enough).

Problem 11: Events (15 points) In the cooperative paper:

1. (2 points) How could their "compute function" `GetCAInfo` cause performance problems with an event or cooperative system?

   Loads and stores can page fault, which means they block on I/O just like `read` *and* `write`*, with the same problems.*

2. (3 points) Why don't they believe it's easy to forget that `GetCAInfoBlocking` has been transformed to continuation passing?

   Signature changed: has a `cont`*. Harder to ignore than silent blocking.*

3. (5 points) What determines how big the memory pointed to by `cont` is? When a thread context switches, does the memory it needs to save state grow at the same rate? Why or why not?

   If you tear code into two continuations C1 and C2 then `cont` *grows with the number of non-global variables defined/written in C1 that are referenced by C2. This is the* *same as a manual context switch: you save the state and then when you switch back, restore it. Memory needed for thread cswitching doesn't grow: it's defined by registers rather than the number of live variables.*

4. (5 points) Which approaches can run `GetCAInfoBlocking` "as is" with no changes and still work? (Note: make sure you state any assumptions.)

   Cooperative, pre-emptive. For the first, you'd have to kick off an asynch I/O and make sure you knew it could yield. For the latter there would likely be locks in the two hash table calls. Also: serial task management.

Problem 12: Superpages (10 points) A program executes the statement `*p = 1` and then:

1. (3 points) The superpage containing the memory `p` points to is demoted. Why? What are the most number of distinct subpages it will produce?

   If it was clean and now dirty it will get demoted. Assuming 4MB superpage: you'll get 7 512K, 7 64K and 8 8K.

   A common mistake was to say it was demoted for memory pressure: but since the page was just written to it was the single most recently accessed page on the system, so it would not have been selected.

2. (3 points) The page containing the memory `p` points to is promoted to a superpage. Why?

   If the containing page was clean and there were enough other dirty pages surrounding it to make a superpage.

3. (4 points) The object containing the memory `p` points is promoted to a `512K` superpage. Give two cases where the reservation could have been 4MB.

   (1) The memory is dynamic: will reserve beyond the end, but only allocate a superpage that is less than or equal to size. (2) the system does not have enough space.

Problem 13: ESX (15 points) Consider Figure 8 in the ESX paper.

1. (5 points) In 8(a) what is strange about the `alloc` line going far above 1GB? How can this happen?

   It's strange b/c they only have 1GB. If two 8K pages get shared the graph might count this as 16K, which would make sense since they say 325MB are shared, giving a number that is roughly that on the graph.

2. (5 points) Consider minute 40 in 8(c) (the citrix server). What is going on with the two peaks and one valley? Which line is driving which?

   Active goes up, making ballooning release memory, making alloc go up.

3. (5 points) At around minute 70 in 8(c) what is going on? Which line is driving which in this case and, in particular, what is the likely reason for `active` going down? (Hint: it may help to look at 8(d).)

   The query SQL query in 8(d) is driving the active line up there, which drives the baloon line down and the alloc line. The effect on 8(c) is that the balloon goes up, pulling the active line down. The active fluctuation is less clear, so we took a variety of answers.