Stanford University
Computer Science Department
CS 240 Midterm Spring 2013

May 6, 2014

!!!!! SKIP 15 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
1-6 (30 points)	
7-12 (30 points)	
13-14 (30 points)	
total (max: 75 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. Your program uses lock-unlock functions Eraser does not know about: what happens? Your program uses a alloc-free functions eraser does not know about: what happens?
   Lock-unlock: You will get false positives everywhere, since locksets will be incomplete. Alloc-free: false positives. The main problem is that eraser must know about re-allocation of memory so that it can reset the locksets associated with the re-allocated memory (since no thread that was using the memory before `free` *should be reusing it and thus intersecting the previous locksets would give false positives).*

2. Give a very short code example where only one thread writes to a variable, but Eraser will report a race. Annotate your figure with any thread switches and mark each line with the exact transition the Eraser state machine is in.
   Code snippets annotated with states that were correct were given full credit. Here is an example:

```
Thread 1                        Thread2
                                                | virgin
   x++;                                         | exclusive
           ---------------->
                                 y = x + 1;     | shared
           <---------------
   x++;                                         | ls = {}, error
```

3. Boehm: as suggested in class, you define the semantics of a `volatile` variable `v` as giving two guarantees: (1) no additional loads or stores can be done to `v` other than what appear in the program text and (2) an access to `v` cannot be reordered with any other volatile access or lock call. Which problems (if any) in Section 4 would this fix?

   It should preclude speculation in loads and stores, which fixes both 4.1 and 4.3. It arguably fixes the global variable problem in 4.2 but we didn't require you say so.

4. If you switch from pre-emptive threads to cooperative threads, which problems (if any) in section 4 would this fix?
   It should fix them all since the code figures do not explicitly `yield()`*. (Note: some lock implementations can yield, but since the compiler has synchronized memory, this will not cause problems.) (Also note: This is for a uniprocessor. If you explicitly mentioned multiprocessor and said it doesn't fix anything in that case, that got full credit too.*

5. After reading the Events are Bad paper, you rip all the events out of your server, and replace them with threads. It works great until a bunch of requests come in that require disk I/O and then your memory usage explodes. What happened? How would you (perhaps partially) fix it?
   If you use standard threads with the standard approach of a large stack limit (e.g., 2MB), then each blocked thread will consume significant memory. Under high load this

can add up. You could use the compiler technique mentioned in the Brewer paper to bound stack size, or use Mesa's approach and explicitly allocate each stack frame on procedure call and free on return (this would require compiler or hardware support :). If you were willing think hard about scheduling and bounding worst-case latencies you could also run another thread on a blocked threads stack.

6. Livelock: In *at most 40 words* and ignoring any speed hacks, give the **complete** livelock solution that they implemented in their system (ignore quotas and grammar). **Please write the word count by each line of your description.**
   About 30 words:
   (a) *After packet arrives, leave interrupts disabled on that interface; re-enable when all packets processed.*
   (b) *Ensure network does not starve application or other interfaces.*
   (c) *Disable input processing when application queue full; re-enable after timeout.*

7. Livelock: Figure 5: Why does "Polling (no quota)" drop almost immediately to zero rather than gradually decreasing similar to "unmodified"? Be very concrete.
   When packets arrive at a sufficient rate, transmit does not get to run: without a quota the receiver will keep going as long as there are packets sufficient packets. The longer path length makes this more likely to happen.

8. "Because memory is plentiful there should be no difference." Give a reasonable question about the superpage paper that would have this as an answer.
   Possible: if you did promotion eagerly, how would Table 1 change? Perhaps: if you ran the benchmarks concurrently, what would happen to Table 1?

9. Rectangle A states that the superpage guys should have measured the increase in memory footprint from using superpages. Rectangle B states any difference should be negligible. Who is more correct and why?
   The superpage guys only promote when a region is fully populated (which also implies the superpage does not extend beyond this region) so they should never use more memory. They also demote under memory pressure in a way that shouldn't increase memory usage.

10. ESX, Figure 8: around the 68 minute mark: explain the causal connection between alloc, balloon, and active in (c) and (d). (I.e., which one is driving the others, and the order in which the others influence each in turn.)
    A long running query is kicked off in (d): (1) this drives active up, (2) the balloon in (d) deflates, allowing active to increase (3) slightly after (it appears) the balloon in (c) starts to inflate, driving alloc down (4) since alloc(c) is down, active(c) also has to decrease since there is less memory to touch.

11. 0000 tells you that an implicit page fault where ESX has invisibly evicted a page and must later fetch it from disk in response to a guest OS reference hurts performance much more than an explicit page fault where a guest OS has explicitly evicted a page

to disk and later has to fetch it back into main memory. 1111 tells you that this is complete nonsense because Carl, who stood to make money from this fact, never mentioned it. Who is right?
You expect invisible page faults to hurt throughput much more than visible ones. For invisible faults, ESX will have to block the guest OS completely since it has no visibility inside of it to run a different kernel thread or process. In contrast, with explicit faults, the guest OS can just switch to another process. (If you just mentioned double paging from the paper, you were given partial credit.)

12. Give three *clear, obvious* examples drawn from the papers we have covered of observability games (these examples might not be contributions of the papers themselves). I.e., examples of doing X instead of Y because code cannot tell the difference.
Concurrency: critical sections. VMware: lying that you are running on the underlying machine. The notion that a program runs sequentially when in fact it's interlaced. compiler optimizations: code reordering, etc. (If all three of your examples were examples from section 4.1, 4.2, 4.3 of Boehm paper, you were given partial credit.)

Problem 13: Mesa (15 points)
Consider the memory allocation code in the Mesa paper.

1. The paper states this code has a bug. What is it and what is the fix?

   Need to broadcast since the woken up allocating thread probably does not need exactly the number of bytes freed and thus other threads should get a chance to allocate what it cannot use.

2. Intuitively, how would you change this code to work with Hoare wakeup semantics?

   It's ugly. The thread woken up in allocate would have to signal if it has any bytes left over (either b/c it needs more or uses less). Plus the woken thread has to check the bytes available in any case.

3. What happens if we make `Expand` an `ENTRY` routine?

   Deadlock.

4. Give an example monitor invariant for this code.

   Monitor invariants are just correctness invariants for the monitor's data structures. So, for example that `availableStorage` *equals the exact number of bytes available, that the freelist holds all and only freed blocks, etc.*

5. Give a quick intuition for where priority inversion could come up in this code.

   Low priority in Free. Interrupted by high priority trying to Alloc. High priority put on lock queue since lock already held. Middle priority thread starts running, preventing Low from making any progress and releasing monitor lock.

Problem 14: Native Client (15 points) Consider the code in figure 3:

1. Give two instructions `inst_is_disallowed` would check for.
   Changing segment registers, system call instructions, any instruction that fries the CPU.

2. Based on the code: do direct jumps have to be aligned to 32-bytes? Do they end the 32-byte blocks?
   You know where direct jumps go, so they don't have to be aligned; therefore they don't have to end a 32-byte block.

3. What is the attack if you delete the check `Block(StartAddr[icount-2] != Block(IP))`?
   Put the first instruction of a nacl jump at the end of one block (B1) and the indirect jump in the next (B2). Jumping to B2 will skip the alignment instruction.

4. For the check `Block(StartAddr[icount-2] != Block(IP))` what is the opcode for the instruction `IP` points to?
   `jmp`.

5. What happens if you delete the four characters: `else`

   Similar to previous question: JumpTargets will contain indirect jumps as valid targets. A direct jump can jump right to them.