Stanford University
Computer Science Department
CS 240 Midterm Spring 2013

May 14, 2013

!!!!!! 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 (45 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. Eraser: Describe an error that the state machine in Figure 4 will detect, but the text that discusses the state machine (which disagrees with the figure) will not. State your intuition and reasoning!

   Bug in the description. The figure is right. Looking at the later description of the implementation, any write will take it to shared-modified. Once it is shared it is running the lockset algorithm without giving warnings, which means that the per-variable shadow area contains the lockset pointer, so it can no longer be keeping track of the thread number of the original writer. We can also reason from what it should do. If anyone is writing into a variable that at least one other thread has been reading from, we have a possibility of a race, so we had better be raising alerts if the locking protocol is violated.

   So: code where the original writer writes again after a new thread has read.

2. You run:

   ```
   eraser ./a.out
   ```

   If eraser emits no error messages, does this mean `a.out` has no errors? If it emits an error, does `a.out` have errors? If you then rerun `a.out`, will eraser emit the same errors? Concisely state why or why not, especially for the last question.

   Does not mean no errors: will have missed almost all paths in the code, doesn't check that the critical section is big enough. Does not mean has errors: has false positives such as not detecting semaphores or happens-before type guarantees (e.g., a thread not forked yet cannot have a race with it's parent thread). Won't emit same errors necessarily since code can be non-determistic (not take same paths, not access same addresses, or have different thread switches therefore affecting initialization differently.)

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. Boehm: In what way does Figure 3 undercut the entire point of section 5.1? (Please state what this point is.)

   They want to show that fancy lock-free synchronization gives performance benefits but the experiments show that you get about the same performance with a single-threaded implementation (which is much easier to reason about).

5. Livelock: Will screend be more or less susceptible to livelock when running on a guest OS on VMware? Use any relevant data from the "Comparison of Virtualization..." paper to support your argument.

   The guest will effectively run slower, so will be more susceptible to livelock. From the comparative paper, the best case slowdown for this type of system-intensive program seems at least 30%.

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. An implicit but overriding principle of the superpage paper is *primum non nocere* ("first, do no harm") in that they try to never be worse than the base system. Give two examples of choices they made that satisfy this principle and one example that does not.

   Many places. Some: (1) they only promote to a superpage when the entire region is populated, (2) they do not do relocation. One place where they don't: they evict pages to restore continuity which can interfere with approximate-LRU.

8. 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.

9. ESX invisibly takes MPN1 from guest OS1; later, a user process running on OS1 writes to a virtual address that maps to MPN1. OS2 takes PPN2 from a user process, which later writes to a virtual address that maps to PPN2. Will OS1's throughput degrade more than OS2's, or vice versa? Please explain your reasoning.

   You expect OS1's throughput to go down more than OS2s. The problem with OS1 is that 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 OS2 can just switch to another process.

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.

Problem 11: 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?

   Needs a loop. Also a 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 allocator 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.

   This isn't a very good question. We had to take most answers.

Problem 12: A Comparison of Software and Hardware Techniques for x86 Virtualization (15 points)
Consider the `isPrime` code in Section 3.2:

1. What does the code for `isPrime` look like when run on their hardware VMM?

   It's unchanged. That's one of the great things about HW virtualization.

2. Judging by the resulting translation did `isPrime` run as unprivileged user code or as privileged kernel code?

   Must have been privileged or it wouldn't have been translated.

3. Give one instruction in the rewritten version of `isPrime` that was produced by self-modifying code.

   The jumps where the JCC comment has been eliminated. When they link to the TC location they go and rewrite the code.

4. Give concrete inputs to eliminate the jumps to `takenAddr` and `fallthrAddr3`.

   Need something less than two and something that is a prime. So `isPrime(1)` *and* `isPrime(3)` *should do it.*

5. Assume `isPrime` is only called from a single location l and, thus, they replace the return instructions in `isPrime` with a hard-wired jump to the instruction after l. Would this optimization (1) always work, (2) sometimes work, (3) never work?

   Sometimes work. It wouldn't "work" (in the sense that it would differ from what a real machine would do) if the return address was corrupted on the stack. We gave partial credit if you mentioned something about having to invalidate jumps if the block they jumped to was evicted, but that's true even without this change.

Problem 13: 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. From 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. 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.

5. If you compute `StartAddr - JumpTargets` what do you get?

   Just the indirect jump instructions.