Stanford: CS 240 Midterm Spring 2019

May 23, 2019

!!!!!! SKIP 15 POINTS WORTH OF QUESTIONS. !!!!!

This is an open-book exam. You have 80 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
1-10 (50 points)	
11-13 (45 points)	
total (max: 80 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. Boehm/Eraser: You see the following code:

   ```
   if(!p) {
     lock();
     if(!p) {
       struct foo *t = foo_new();
       t->bar = bar_new();
       p = t;
     }
     unlock();
   }
   ```

   Eraser said this "double-check lock" idiom is broken because of memory consistency. Assume the hardware is sequentially consistent and Boehm fixed the compiler to not be stupid about locks. Does this code still break? Why or why not?
   Still breaks. The code assumes that `p` *is only set to non-null after the field* `bar` *has been initialized, but the compiler can get rid of the temporary assignment to* `t` *and just assign* `p` *immediately. If so, another thread could then read* `bar` *before it was set to anything.*

2. MESA: Your cs140 partner states that because of the way that Mesa's scheduler works, you can implement locks by just raising your thread's priority to the highest level. Give two examples where this approach *could* work better than the original MESA implementation. Will it work correctly sometimes, never, always? Explain.
   This could sometimes work. If there was only one monitor and it didn't block, perhaps it would be better because setting a priority could be cheaper. In addition, you could maybe see it as a priority donation method where any thread "holding a lock" had a higher priority than others. Blocking in nested monitors would definitely be a problem.

3. Eraser/Boehm: your cs140 partner says you can catch when the compiler violates `pthreads` assumptions by compiling the checked code once optimized and again unoptimized and comparing the difference in Eraser results on both binaries. What's the intuitive argument here? For the bugs in the Boehm paper (sections 4.1, 4.2, 4.3), what will

happen?
The intuition: the problem in the paper was the compiler would optimize code assuming only the code itself could observe the values rather than the (correct) assumption that other threads could observe these as well. For bugs bug caused by the compiler doing speculative code motion based on this faulty assumption comparing the unoptimized and optimized diffs (assuming deterministic code) should often point this out. 4.1 and 4.3 should show up. The non-atomic writes in 4.2 would remain the same in both for the most part (though the speculative large write might show a difference).

4. You have an intern that reads the livelock paper and changes MESA's networking module to not use interrupts but instead has a single thread that stays in an infinite loop, calling the following routine (`wait_for_pkt`) to get packets. The network device signals the condition variable using the "well-known wakeup-waiting switch" discussed in MESA.

```
condition_var has_packet;
ENTRY wait_for_pkt() -> packet BEGIN
   // if network interace has no packet, wait.
   while(nic.empty())
      wait(has_packet);
   // pull packet off the nic, return it.
   return nic.get_packet();
END
```

   If you run the test in Figure 5 on this system, which line will its output likely be closest to? (Explain why.) What about for Figure 6? (Explain why.)
   The code switches from a "push" architecture to a "pull," so it has that improvement. The use of a binary semaphore means it doesn't have the lost wakeup problem. However, as described, there is no quota, so it should follow the "no-quota" line. Also it has no way to prevent starvation, so should follow "polling, no feedback" the closest (we also took "unmodified").

5. Livelock, Figure 7: given the trends for different quota sizes, it sure looks like a quota of 1 would give the best performance. Use data from the paper to argue whether this belief is true or not.

Figure 8 computes the optimal point (though there are no error bars), which gives closer to 8 — a quota of 1 is clearly worse, at least in this experiment. (Personally I would like to see something more than a single max value.)

6. Superpages: LRU-BSD claims to modify stock BSD (i.e., with no superpages) so that physical pages get re-allocated using guaranteed, perfect LRU. You re-run the experiments in Table 1 on top of LRU-BSD on top of the same hardware. You notice `eon`'s runtime is about 20% worse than when run on supepages. What would the superpage people say is going on given the data?
   If we just take pages in LRU order, we're not obeying page coloring, this means we can get more cache conflicts. Superpages, by its nature, does page coloring which — as discussed in class — can get about 20% (or more, or less depending on the program / hardware). We know for sure it's not TLB related, since `eon` *had neglible TLB issues.*

7. If you run the superpage BSD system on top of ESX, why might it get no (or negative) improvement over BSD without superpages? How can you solve this problem with modifications in ESX, without affecting non-superpage mappings?
   It'd be worse b/c ESX didn't do anything about superpages. So the guest OS could try to use them, but the contiguous PPN's it's using would likely map to scattered MPNs. ESX could try to preserve continuity, but that could harm guest OSes that aren't using superpages (every time you try to optimize for an additional variable, typically you do worse on the other variables). ESX could align MPNs with PPNs for ranges in guest OS page tables used for superpages. It could also initially align MPNs and then back off if they weren't being used in superpages.

8. ESX takes a machine page from guest OS1. Guest OS2 takes a physical page from a user process. Under identical workloads would you expect OS1's throughput to degrade more than OS2's, or vice versa? Please explain your reasoning, referring to other papers to support your reasoning.
   From ESX's point of view the guest OS is largely a black box. ESX has no way to "switch threads" in the guest. Thus, if a guest faults on a PPN that is paged out, the whole guest is blocked. In contrast, a

guest OS can easily switch to another process if one process blocks on a paged-out PPN.

9. Microsoft constantly gets blamed when a third-party device driver corrupts state and causes the OS to crash. Sketch how to adapt NACL to handle this, assuming that we cannot use segmentation. You can assume (dubiously!) that x86 has enough registers.

   Make sure to mention how to: modify the NACL idiom, handle loads and stores, any alignment issues, and where to change the NACL verifier code.
   This is a pretty big question. We accepted the high level points:

   (a) Without data segmentation, you have to smash the upper bits of every load or store to be within a contiguous range. Thus, you'd have to use a "nacl load/store" that did a bitwise "and" to clear the upper bits, and a bitwise "or" to set them.

   (b) Same as above for jump: nacl jumps would have to modify the upper bits as well.

   (c) These sequences would be longer than a nacl jump, so you might need larger blocks (e.g., 64- alignment vs 32) since they can't span the block size.

   (d) You'd have to extend the nacl jump check in the verifier to handle the larger jump sequence and also to handle the nacl load/store. In particular, you have to check (1) thse don't span blocks and (2) all instructions needed cannot be direct jump targets.

10. ESX, Figure 7: given the alleged tax rate, solve Carl's formula to determine the equilibrium share of pages for A and B if A is 100% busy, and B is 100% idle. Does the figure indicate Carl's implementation is broken? Why or why not?
    It should be a 4:1 ratio:

```
     S/(P (f + k (1 - f)))
   = S/(P (f + (1 - f) / (1 -r))
   = (f + (1 - f) / (1 - .75))
   = (f + (1 - f) / .25))
   = (f + 4(1 - f))
   = (4 - 3f)
```

```
for 100 = 4 - 3 * 1 = 1
for 0   = 4 - 3 * 0 = 4
```

Since shares are equal, one process should get 4x more than the other. ESX splits up 360MB b/n two VMs capped at 256MB. We'd expect 90MB vs 270MB after the 32ish minute mark when the tax rate is raised to 75%, but since they capped the VM quota to 256MB, that's about what they converge to.

Problem 11: Eventful (15 points)

1. (3 points) Behren, Events-bad, Figure 3: explain from a livelock perspective what is going on. How does their explanation differ?
   If you favor accept, you'd expect throughput to go down under high load. They don't even really mention this — they say its related to context switching, many small modules, and many temp objects. Could be this, or could be livelock-ish. Hard to say! (Now perhaps you appreciate the livelock throughness more.)

2. (3 points) Atul, Cooperative: How is their definition of a "compute function" naive?
   If it has a load or a store, those could block.

3. (3 points) Atul, Cooperative, Section 3: State all the lines in the stack ripped code that correspond to context switching in threads.

   The lines that allocate state, save it, and load it.

4. (3 points) What happens if a procedure `foo` calls `GetCAInfoBlocking` in the middle of its code and you turn `GetCAInfoBlocking` into an event handler?
   You have to rip `foo` *as well (and its callers).*

5. (3 points) Give the one bullet in the Outsterhout talk that does the most to invalidate the title.
   Threads are pre-emptive.

Problem 12: ESX / Eraser (15 points)

Carl thinks the concurrency property checked by Eraser is too strong and too weak. Instead he wants to make a product for checking library code, where he will run two routines back to back "A; B" and "B; A" and record the result for both, then rerun do context switching at every instruction.

(Hint: Recall our long discussion about how to check that different computations have the same result.)

1. (10 points) Explain how to use ESX binary rewriting and copy on write abilities to do this.
   You can build a snapshot method using COW to quickly snapshot the memory state before running A;B. You can record the state after. Then reverse the snapshot and do `B;A`*. Then do interleavings (using binary rewriting) on each instruction.*

2. (2.5 points) Give a "as if" argument for why he can switch much less often than every instruction and be guaranteed to find the same errors.
   Only the loads or stores matter. Just switch on these.

3. (2.5 points) Give a specific example from the Eraser paper where he could potentially have better results and one false positive that he would still have trouble with.
   He's still have the false positive about the statistics variable. He could pass a lock free data structure implementation that would fail otherwise. He could detect cases where critical sections are not big enough that eraser would miss.

Problem 13: Observability games

We discussed different tricks you can do if A behaves "as if" it was B. One of our main examples were compiler optimizations, where the common compiler rule that you can replace code A with code B if you can guarantee that running A on the same initial memory state must produce the same final memory state as B. Answer the following questions using intuitions about observability.

Assume each code snippet is the only thing in the the file and that the compiler only looks at one file at a time and does not use any auxilary inter-file information.

1. Which writes can the compiler remove (if any)?

   ```
   static int x;
   void bar();

   void foo(int *p) {
       x = 1;
       bar();
       x = 2;
       return;
   }
   ```

   Remove `x` *entirely and all its writes. Nothing can observe it.*

2. Based on observability, can the compiler re-order or remove any of the following writes? Why or why not?

   ```
   void foo(int *p, int *q) {
       *q = 1;
       *p = 2;
       *q = 3;
       return;
   }
   ```

   The compiler can't know if `p` *and* `q` *point to each other, so can't reorder (unless it inserts a dynamic check). It can remove the first store. Note: we might have incorrectly taken points off if you said you could eliminate the first store — please let us know if so. Also, if* `p` *points to* `q` *the second write would* `q` *— this isn't conformant behavior, but we would have given full points.*

3. How much of this code can `gcc` remove?

   ```
   #include <stdlib.h>
   int main(void) {
       int *p = malloc(4);
       *p = 10;
       return 0;
   }
   ```

   All of it besides the return. Try it!

4. Assume `nic->status` is memory-mapped location, that will return a 1 when a load occurs if the network interface (NIC) has data, and 0 otherwise. Based on our class discussion, what can happen with this code? How to fix it?

   ```
   void wait_until_ready(nic *n) {
       while(!n->status)
           ;
   }
   ```

   Infinite loop. The loop does not write the location, so the compile will check `status`*: if its not 0, then it never does the loop, if it is, it makes the loop infinite. Fix:* `volatile nic *n`

5. Assume we are trying to measure the cost of a load. What are two ways this code could return 0? How to fix these issues?

   ```
   // assume reading from 0x20003004 gives the current
   // time in nanoseconds.
   int ld(unsigned *time, int *p) {
       unsigned start = *(unsigned *) 0x20003004;
       int l = *p;
       unsigned end = *(unsigned *) 0x20003004;

       *time = end - start;
       return l; // so compiler does not remove.
   }
   ```

   Removes the start and end and just writes 0. Moves the load outside of the two measurements.