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
1-6 (30 points)	
7-12 (30 points)	
12-16 (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. 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?

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.

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?

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.)

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.

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?

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?

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.

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.

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?

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?

2. (3 points) Atul, Cooperative: How is their definition of a "compute function" naive?

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

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?

5. (3 points) Give the one bullet in the Outsterhout talk that does the most to invalidate the title.

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.

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.

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.

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;
}
```

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;
}
```

3. How much of this code can gcc remove?

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

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)
        ;
}
```

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.
}
```