Stanford University
Computer Science Department
CS 240 Midterm Spring 2020

October 25, 2024

!!!!!! 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
1-10 (50 points)	
11-13 (45 points)	
total (max: 75 points):	

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

1. As mentioned during office hours: one approach to running cooperative threads concurrently is to "color" them by the memory they use. If two threads use only disjoint memory locations, they are given different colors. Threads with different colors can run concurrently. Of course, mistakes will be made, with sad results. How could you extend Eraser to detect coloring errors? (Make sure to explain how memory reuse works compared to the unaltered Eraser.)

2. You run the following code on the Alpha computer in the Eraser paper:

```
char a,b;

     T1   |          T2
lock(a_lock);         |         lock(b_lock);
a += 1;               |         b += 1;
unlock(a_lock);       |         unlock(b_lock);
```

What will Eraser do? Is it correct? What does Boehm say about this type of code and POSIX?

3. Your cs140 partner wants to make CloudEraser, which will attempt to aggregate Eraser checking runs for a given binary done anywhere in the world. Their plan is for each client to send a central server the lockset, thread id, and variable address for each load and store performed in the program, along with a hash of the program (to identify it).

   Let's stipulate their scheme works: could it find significantly more errors over a single Eraser run? Why or why not?

   Ignoring security concerns, give a specific technical problem Eraser addresses that the above approach, at least as sketched, will have trouble handling.

4. Boehm states compilers typically do not understand POSIX `lock` and `unlock` and try to guarantee they are handled "appropriately" by simply treating them as opaque procedure calls.

   However, your compiler also has a fancy (somewhat common) optimization trick where it says if a variable `v` does not escape the current scope, it knows opaque functions cannot read or write `v` and so can reorder the call/variable access. For example in the code:

```
// begin file foo.c
static int x;  // name not visible outside foo.c
int foo(void) {
     x++;
     bar();   // compiler decides bar() cannot read/write x.
     return x;
}
// end file foo.c
```

   The compiler can decide `bar()` can't read or write `x` (since the address of `x` is not passed anywhere and `x` is statically scoped and cannot escape the file) and then reorder these statements.

   How can this optimization interact badly with the lock rule above? Give a three-line example.

5. Adya, Cooperative: Your lab partner claims they make a program that correctly uses cooperative threads to correctly run using pre-emptive threads by using the following strategy:

   (a) Having a single global lock `L` for the entire program;

   (b) Before any thread runs, it acquires the global lock `L` if it doesn't already hold it;

   (c) Before any thread explicitly blocks it releases lock `L`, (but does not release it when pre-empted).

   What is the intuition from Adya's paper behind this transformation? Ignoring deadlock concerns and assuming the thread's package works with this locking approach: will code that was correct and race-free when run cooperatively still be correct and race-free in the above scheme? Why or why not?

6. A random Zoom chat claims you can implement MESA's `signal()` and `wait()` for a modern `pthreads` with the following type signatures:

```
void signal(cond_t *condition);
void wait(cond_t *condition);
```

   Will this interface work for cooperative threading? Why or why not? Will it work for pre-emptive threading? Why or why not?

7. Livelock: assume you are running two network user-level server applications concurrently instead of one, where the second also produces one outgoing packet for each incoming packet. If we redo the `Figure 6`'s experiments for (`screend`) and re-graph the results, what are two potential significant differences we could see for the `polling (no feedback)` and `polling (feedback)` lines? Make sure you give the intuition for your answer.

8. Livelock: the network interface (NIC) makers read both the livelock and the NACL papers and decide to allow the OS to download code and some amount of data onto the NIC to run on each packet. Sketch how to use this ability to build a better solution than is in the livelock paper? Give a specific workload where you would **obviously, clearly** do better than the livelock paper. For easy of answering: You can assume the NIC can access host memory.

9. Superpages: Jimbob says the most important thing to know about the future is when an object will die. Bobjim says that the most important thing is how an object will be populated. How could you use these facts? Assuming the experiments in the paper are a good test of real workloads, give an experimental argument for who is more right.

10. When VMware was founded, most OSes didn't support superpages. VMware hires Navarro to hack ESX so that ESX will trick guest OSes into using superpages. At a high-level: how would he do so? What decision made in the superpage paper will you likely have to change to make this approach get any benefit and why?

Problem 11: NACL (10 points) It's not atypical for 90% of an OS to be device driver code, which is typically not-great quality and, when it runs unprotected inside the kernel, can easily corrupt kernel memory. Since end-users blame the OS for all crashes, no matter who caused them, when some far-sighted thinkers at Microsoft see cs240 on your resume they hire you to build a variation of Nacl to protect the OS against drivers.

In broad strokes, what what ideas can you use relatively unchanged, and what new problems will you have to solve? State any major assumptions you have to make. Make sure you address: (1) the lack of system calls, (2) the fact that both the driver and OS run in the same address space, and (3) that a crashed driver may be using kernel locks or allocated kernel memory.

You don't need to write pages of prose; just the intuition behind same/different and intuitive sketches of techniques are fine.

Problem 12: ESX (10 points)

ESX: Assume we run an experiment similar to Figure 8, which has the most strong mirroring / reflection between experimentally derived lines of any Figure I can think of in systems. Explain what kind of workload and machine conditions you would need to break the mirroring in the following contexts:

1. A guestOS's `alloc` and `active` lines increase, and mirror each other, but its balloon line stays flat at zero.

2. Do you expect `alloc` or `active` to be slightly right shifted?

3. (2.5 points) A guestOS's `active` line increases dramatically, but `alloc` actually goes down.

4. (2.5 points): A guestOS's `balloon` line remains the same but its `alloc` decreases dramatically.

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?

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

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

4. What happens if we make the `wait` call just put the current thread at the end of the run queue?

5. Give the main monitor invariant for this code.