Stanford University
Computer Science Department
CS 240 Midterm Spring 2020

May 15, 2020

!!!!!! 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.)

   Should be a simple extension: we want to check that different thread colors never touch the same address.

   (a) Initially set memory to a virgin state.

   (b) When a thread accesses virgin memory, set the memory to the thread's color.

   (c) When a thread accesses non-virgin memory, give an error if it has a different color (which implies non-disjoint memory accesses).

   (d) When memory is freed, reset freed memory back to the virgin state.

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?

   If `a` *and* `b` *are placed in the same machine word, then Eraser will flag*

this as an error — it only tracks a lockset per 32-bit word. The Alpha they ran on did not have store/load byte instructions so this would be a true error. Boehm stated this is a conforming Posix program, and complained that the error introduced above was allowed by 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.

   Eraser could potentially find many more errors. Eraser can't find bugs in paths it does not execute. Thus the most obvious way to find more bugs is if multiple runs execute different paths. Further, it can only find errors on a path with the actual values used at runtime. If two runs have different values that (for example) cause them to access different memory locations, then it could also find more errors.

   A big challange is that Eraser works in terms of addresses for memory and for locks. These can change from run to run. Also, it wants to see when you malloc and free so it can reset the memory state to Virgin to reduce false positives. CloudEraser would have to track this information too, to reduce false reports. However, a naive approach may make it difficult to figure out which "lifetime" of a memory location a lockset corresponds to.

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.

```
lock();
x++;
unlock();
```

can get written to:

```
x++;
lock();
unlock();
```

Since the compiler decides that `lock` *and* `unlock` *cannot access* `x`*.*

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?

As Adya et al note, cooperative threading = by default everything is a critical section, broken up only by blocking/yielding operations. To translate this to pre-emptive, you would map it to a single lock, acquired as soon as you start running (scheduled the first time or unblocked), released when you block. If you run Eraser on the translated copy there should be no warnings (other than those noted by Boehm :), since the thread will always hold a lock (this is similar to aquiring a lock before running an instruction and releasing it after). Should be race-free; won't be that concurrent, however.

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?

You have to guard against the lost wakeup problem:

```
if(<some condition>)
 <----- another thread could send a wakup here.
wait(c);
```

For cooperative, you can't get interrupted, so there is no problem. For pre-emptive, you'd have to pass in the lock used to guard your condition.

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.

   One reasonable point: Running two applications vs one makes the CPU effectively slower, which should reduce the MLFRR for both compared to if they were running solo. Another: if one application's queue fills up, all network processing will be shut off, for up to 1ms. This can cause the other graph to have many points where its throughput drops to 0.

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.

   Run code on the NIC to decide which packet it is for. When the kernel thread gets the packet give it the queue ID. Discard if the queue is full. Probably should do polling; though if the check is fast enough it's not needed.

   We also accepted: Run code to decide which queue the packet is for: if it's full, drop it. Though this method of looking in the CPU's memory for the queue is likely unrealistic.

This should be much better than their screend hack — in particular in the livelock paper's system if you are running many network programs and one gets behind NIC will get shutdown for a 1ms, whereas this system could be more precise and just discard that single apps packets.

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.

   You could use death times for placement: put things that will die together near each other so that death frees up a lot of space. You could use population information to decide on whether to do eager promotion or not. From the experiments, the overhead of lazy promotion and general management doesn't seem to be such a big deal, so probably placement is more important (since it lets you use superpages when memory contention is much higher and the benefit of superpages for many apps is non-trivial.).

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?

    What he would want to do is to map the PPN's of contiguous VPNs to contiguous MPNs and insert this into shadow page tables to they get put in the TLB at miss time. However, this probably won't be that effective: the guest OS will just pick PPNs it wants, which will almost certainly not map to contiguous MPNs. He would likekly have to do some sort of compaction scheme to see any real continuity.

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.

1. *You can reuse the notion of scanning, making code safe, and using trampolines for callouts.*

2. *You may have to modify the kernel to allow the use of segmentation protection.*

3. *You'd have to track all resources used by the driver so that if it dies, you can go through and release them.*

4. *need a way to catch exceptions safely as well as too-long execution.*

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.
   There is plenty of memory, and other processes are idle. The more we actively use, the more we get.

2. Do you expect `alloc` or `active` to be slightly right shifted?
   Ignoring other GuestOS actions, `active` *will drive* `alloc`*, so we expect* `alloc` *to slighly lag and be to the right. We also took: balloon goes up, then alloc goes down, which will drag active potentially.*

3. (2.5 points) A guestOS's `active` line increases dramatically, but `alloc` actually goes down.
   A different, higher priority but previously idle GuestOS's `active` *goes up, so it preferentially gets memory.*

4. (2.5 points): A guestOS's `balloon` line remains the same but its `alloc` decreases dramatically.
   Same as 1. Could also happen if there was forcible revocation.

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?
   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?
   Allocator will have to signal if it has any bytes left over (either b/c it needs more or uses less). Woken up thread has to check the bytes available in any case, so is no better than MESA semantics.

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

4. What happens if we make the `wait` call just put the current thread at the end of the run queue?
   Doesn't matter for correctness: when it's scheduled, it will just recheck the condition and go back to sleep if not satisfied. This may waste cycles of course.

5. Give the main 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.*