Stanford University
Computer Science Department
CS 240 Midterm Spring 2018

May 1, 2019

!!!!!! SKIP 15 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
1-6 (30 points)	
7-15 (30 points)	
16 (15 points)	
total (max: 75 points):	

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

1. You mis-annotate a function `foo(p)` as a lock routine in in Eraser, what will happen? You mis-annotate a function `bar(p)` as an unlock function, what will happen? **Consider the cases where `p` is or is not a lock variable, and where only one vs multiple threads call the functions.**

   (a) *lock: If you annotate* `foo` *as a lock function but only one thread calls it, no difference (there will be an extra false lock in the lockset, but no one else uses it so it's never preserved in the intersection check). If multiple threads call it, can get false negatives since eraser will think you hold a lock.*

   (b) *unlock: if the parameter* `p` *passed to unlock is not a lock, then the unlock will have no effect. Otherwise you can get false positive errors since eraser will think you release* `p` *and thus remove it from the lock set.*

   There were answers that took the question as: if you claimed the variable `p` *was a lock, then Eraser will not check* `p`*, so you can miss errors. We tried to give points depending on how internally consistent this answer was.*

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 how this behaves on Posix Pthreads?

   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. Give a 3-line code example where only one thread writes to a variable and Eraser flags an error. Annotate your figure with any thread switches and mark each line with the exact transition the Eraser state machine is in.

```
Thread 1                        Thread2
                                                  | virgin
   x++;                                           | exclusive
             ---------------->
                                 y = x + 1;       | shared
             <----------------
   x++;                                           | ls = {}, error
```

4. Boehm: your compiler treats lock calls as opaque procedures that can read or write anything and thus may preclude reordering.

   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 them. For example in the code:

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

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. Knot: in terms events, pre-emption, blocking I/O and Mesa stack management, give a concise description of what Knot does to solve the problems the three event/thread papers complain about.

   They use events to dispatch, but each event is a cooperative thread. Blocking I/O is handled by a pool of kernel threads on the side. They do a similar trick as MESA with thread stacks (roughly: allocating each individually vs one big thread stack) to control total size.

6. You have a cooperative threading program P, explain how to translate it to an equivalent pre-emptive thread program in terms of critical sections and locks. When you run Eraser on both, what do you expect to happen?

   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). When you run cooperative if there are no locks you'll get lots of false positives.

7. Some people claim 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.

8. 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 (e.g., using round robin).*

   (c) *Disable input processing when application queue full; re-enable after timeout.*

9. 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.

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.

10. NaCl: assume you are on a machine only has 4-byte instructions, where the machine will throw an exception if code attempts to jump in the middle of any instruction. Explain which restrictions NaCL can drop and remove everything from the verifier code in Figure 3 that is no longer needed.

    We (would have) taken two answers. The first, which no one mentioned, is that since the trampolines to call out of the nacl software require 32-byte alignment to guard against skipped important checks, you still need nacl jumps, and so need much of what's there already — you can eliminate overlapping block instruction checks (other than the nacl jump), but that seems to be about it.

    However, given that we didn't cover this point well in class, we took the other answer of: since you don't care about jumping in the middle of an instruction, you don't need the 32 byte alignment, thus you don't need 32-byte blocks nor the pseudo call code (and don't need to prevent jumping over the `and`*). Thus the verifier is a simple scan from beginning to end looking for disallowed instructions.*

    If you remarked you didn't need some conditions, but didn't realize you could discard almost all of the verifier, we gave some number of partial credit points, depending on how worked out the answer was.

11. NaCl: you want to port NacL to a architecture that does not provide segmentation for data loads and stores. Explain how to extend their

idea for making jumps safe to additionally make loads and stores safe.

Assume the machine only provides one load instruction "`ld r_dst, [r_src]`" (load the value at address `r_src` into `r_dst`) and one store instruction "`st r_src, [r_dst]`" (store the value in `r_src` into the memory given by address `r_dst`) give the specific instructions you would do to enforce all loads and stores accesss a contiguous address range between `0xfff0 0000` and `0xff00 0000`.

This is the trick introducted by the original software fault isolation paper they discuss. You know all addresses in the range will have the same upper bits, thus you can force all loads and stores to refer to that range by doing:

```
# clear upper bits
and tmp, r, 0x00ff ffff
# set upper bits to 0xffxx xxxx
or  tmp, tmp, 0xff00 0000
# load
ld r_dst, [tmp]
```

This will be a no-op for correct programs and, for incorrect, smash the value they load back into the legal range.

One issue is that you have to ensure that a jump cannot skip this code, so you would have to remove these from jump targets, and put them at the beginning of the block. (The original paper had a cute hack where you just ensure that tmp only holds a valid address at the time of the jump.

12. "Because memory is plentiful there should be no difference." Give a reasonable question about the superpage paper that would have this as an answer.

    Possible: if you did promotion eagerly, how would Table 1 change? Perhaps: if you ran the benchmarks concurrently, what would happen to Table 1?

13. ESX: Figure 6:What bad things would happen if ESX used the average rather than the max of the three moving averages of memory usage?

Very bad: if slow moving was low, and an application started using an enormous amount of memory ESX would keep its active set small, causing the application to have way to little memory and do very bad things. Not as probable, but still bad: if the application went through cycles of being busy and then during idle the fast average will go low, dragging the entire average down, causing ESX to take back memory (paging it out, etc) that it will then have to immediately have to give back.

14. ESX: give a spot on one of the performance graphs where you are most likely to see the serious problem ballooning was trying to guard against.

    If ballooning is working, then we shouldn't have double-paging issues; it's when ESX has to forcibly revoke memory that we get the problem — Figure 8 at the beginning, when it goes into the low state it's doing a significant amount of revocation so is the most likely place to see this.

15. ESX, Figure 8: If your guest OS is losing memory, explain the strongest causal connection between alloc, balloon, and active in terms of how they interact just within your guest OS. If your guest OS is gaining memory, explain the strongest causal connection between alloc, ballon, and active between yours and other people's guest OSes.

    If you are losing memory, then your ballooning is going up, and then your real allocation will go down. If you are getting memory, your active went up, someone else's balloon went up, your balloon went down, and your alloc then went up.

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.*