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)	
7-12 (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. 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.**

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?

3. Give a 3-line C 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. (We don't count type declarations against your 3-line budget)

```
Thread 1            Thread2      | State transitions.
                                 |
                                 |
                                 |
                                 |
                                 |
                                 |
                                 |
                                 |
                                 |
                                 |
```

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.

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.

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?

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?

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.**

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.

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.

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`. (You may assume the use of a temporary register.)

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.

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?

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.

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.

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.