Stanford University
Computer Science Department
CS 240 Midterm Spring 2012

May 5, 2012

!!!!! 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)	
13-15 (25 points)	
total (max: 70 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. A thread runs the following procedure:

```
void foo(void) {
        int flag = 0;
        do_work(&flag);
        ...
}
```

   Later on in the program after all threads have exited Eraser flags an error in the code:

```
void fact(int n) {
        return (n<=1) ?  1 : n * fact(n-1);
}
```

   What is going on? (Hint: don't think too much about the code inside of `fact`: it's what you don't see that is important.)

   Underlying problem: eraser will not mark procedure calls as allocating and freeing memory. Therefore, stack variables are never reset. In the specific code, assume: (1) other threads access `flag`*, (2) thus eraser will start tracking it, (3)* `flag` *gets put in shared-modified state, (4)* `fact` *reuses the same stack memory later, (5) eraser will flag an error.*

2. Eraser: what is the contradiction between Figure 4 and the text? Which is correct?

   Oops, a bug in the description. The figure is right. Looking at the later description of the implementation, any write will take it to shared-modified. Once it is shared it is running the lockset algorithm without giving warnings, which means that the per-variable shadow area contains the lockset pointer, so it can no longer be keeping track of the thread number of the original writer. We can also reason from what it should do. If anyone is writing into a variable that at least one other thread has been reading from, we have a possibility of a race, so we had better we raising alerts if the locking protocol is violated. [a legalistic reading of the text can claim that it is technically accurate; it is true that a write access from a new thread in the Shared state does take it to the Shared-Modified state; they just didn't bother to mention that a write access from the old thread in the Shared state also takes the variable to the Shared-Modified state. Under that interpretation the sin is that the authors forgot to mention one important case.]

3. You run:

```
eraser ./a.out
```

If eraser emits no error messages, does this mean `a.out` has no errors? If it emits an error, does `a.out` have errors? If you then rerun `a.out`, will eraser emit the same errors? Concisely state why or why not, especially for the last question.

Does not mean no errors: will have missed almost all paths in the code, doesn't check that the critical section is big enough. Does not mean has errors: has false positives such as not detecting semaphores or happens-before type guarantees (e.g., a thread not forked yet cannot have a race with it's parent thread). Won't emit same errors necessarily since code can be non-determistic (not take same paths, not access same addresses, or have different thread switches therefore affecting initialization differently.)

4. We missed an important point in class: what is weird about how Mesa allocates procedure call records? How does this help bound the storage used for thread stacks?

   They do heap allocation of stack frames (in hardware!). This allows them to more tightly bound thread stack usage by incrementally adding it as needed rather than allocating one huge stack for each thread which will be either too large or too small.

5. You like `signal()` and `wait()` but don't want to write code in a dead language, so build an implementation in `pthreads` with the following type signatures:

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

   Your cs140 partner says you're going to have problems if the only parameter they take is a condition variable. What else would you have to pass in? If you already standardized your interface what implementation hack could you use from the Mesa paper to make things work (possibly)?

   This interface guarantees a lost wakeup. Assume a thread does:

```
if(<some  condition>)
        wait(&c);
```

   The problem comes up when: (1) thread 1 checks the condition, which is false, (2) this thread gets interrupted between the if-statement and the wait, (3) another thread then signals `c`. *Thread 1 will miss the signal and block. You can fix the problem by (1) requiring both threads hold a lock (i.e., manually emulating a monitor), (2) passing the lock into* `wait` *(which will release it after putting the thread on the condition variable's queue). You could also make condition variables "sticky" (ala the wakeup-waiting switch in MESA) and require they do* `while` *loops instead of* `if` *statements to check conditions.*

6. Boehm: You glance through the `NFW-threads` standard and notice the sentence: "A standard conforming program must use exactly one lock." Which problems (if any) in Section 4 of the Boehm paper will this eliminate?

It should fix the problem in 4.2, where a user protects fields `a` *and* `b` *in the same structure with different locks but the underlying hardware will write both* `a` *and* `b` *simultaneously. If must always hold one lock this clearly cannot happen. It doesn't seem to help any of the other sections.*

7. Boehm: as suggested in class, you define the semantics of a `volatile` variable `v` as giving two guarantees: (1) no additional loads or stores can be done to `v` other than what appear in the program text and (2) an access to `v` cannot be reordered with any other volatile access or lock call. Which problems (if any) in Section 4 would this fix?

   It should preclude speculation in loads and stores, which fixes both 4.1 and 4.3. It arguably fixes the global variable problem in 4.2 but we didn't require you say so.

8. Give two places where scheduler activations block without notifying the user.

   If you take away the last physical processor, take a page fault in the thread scheduler, or complete a thread which blocked in the kernel.

9. An implicit but overriding principle of the superpage paper is *primum non nocere* ("first, do no harm") in that they try to never be worse than the base system. Give two examples of choices they made that satisfy this principle and one example that does not.

   Many places. Some: (1) they only promote to a superpage when the entire region is populated, (2) they do not do relocation. One place where they don't: they evict pages to restore continuity which can interfere with approximate-LRU.

10. Rectangle A states that the superpage guys should have measured the increase in memory footprint from using superpages. Rectangle B states any difference should be negligible. Who is more correct and why?

    The superpage guys only promote when a region is fully populated (which also implies the superpage does not extend beyond this region) so they should never use more memory.

11. In what way does ESX's transparent page revocation make guest OSes have a drawback of user-level threads? How could you modify the ESX/guest interface to mitigate this problem?

    Assume: (1) ESX transparently revokes a page `P` *from a guest OS (2) a process or kernel thread in the guest OS references* `P`*. ESX will have to block the guest completely since it has no visibility inside of it to run a different kernel thread or process. You could steal ideas from the scheduler activation paper and do upcalls into the guest telling it about these blocking events.*

12. ESX, Figure 8: around the 68 minute mark: explain the causal connection between alloc, balloon, and active in (c) and (d). (I.e., which one is driving the others, and the order in which the others influence each in turn.)

    A long running query is kicked off in (d): (1) this drives active up, (2) the balloon in (d) deflates, allowing active to increase (3) slightly after (it appears) the ballon in (c) starts to inflate, driving alloc down (4) since alloc(c) is down, active(c) also has to decrease since there is less memory to touch.

13. Will a guest OS be more or less susceptible to livelock when running on VMware? Let's say you are running screend on the "unmodified" OS from the livelock paper and that VMware knows you are forwarding packets. What could it do to detect and prevent livelock?

    The guest will effectively runs slower, so will be more susceptible to livelock.

    If it knows that the guest is doing packet forwarding it could monitor what the input-output rate is and if it drops below the MLFRR point start discarding packets before the guest sees them.

14. Your system has two kernel threads A and B with a message queue Q between them. There is also a user level process C. Assume the system is under heavy load and does not use the techniques from the livelock paper: give two bad things you would expect to see. In *at most 40 words* give the give the **complete** livelock solution for this system (ignore quotas and grammar).

    Possible bad things:

    (a) Messages get dropped on Q.

    (b) B, C starve.

    (c) No output.

    About 30 words:

    (a) After packet arrives, leave interrupts disabled on that interface; reenable when all packets processed.

    (b) Ensure network does not starve application or other interfaces.

    (c) Disable input processing when application queue full; reenable after timeout.

Problem 15: Native Client (15 points) Consider the code in figure 3:

1. (3 points) Give three instructions `inst_is_disallowed` would check for.

   Changing segment registers, system call instructions, any instruction that fries the CPU.

2. (3 points) From the code: do direct jumps have to be aligned to 32-bytes? Do they end the 32-byte blocks?

   You know where direct jumps go, so they don't have to be aligned; therefore they don't have to end a 32-byte block.

3. (3 points) What is the attack if you delete the check `Block(StartAddr[icount-2] != Block(IP)`?

   Put the first instruction of a nacl jump at the end of one block (B1) and the indirect jump in the next (B2). Jumping to B2 will skip the alignment instruction.

4. (3 points) What happens if you delete the four characters: `else`

   Similar to previous question: JumpTargets will contain indirect jumps as valid targets. A direct jump can jump right to them.

5. (3 points) If you compute `StartAddr - JumpTargets` what do you get?

   Everything besides indirect jump instructions.