Stanford University
Computer Science Department
CS 240 Midterm Aut 2024

November 7, 2024

!!!!!! SKIP 15 POINTS WORTH OF QUESTIONS. !!!!!

This is an open-book exam. You have 80 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
1-11 (55 points)	
12 (10 points)	
13 (10 points)	
total (max: 60 points — skip 15):	

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

1. We don't need binary rewriting to detect many threading problems. Explain how you can change a threading library's code to detect both (1) if cooperative threads run "too long" and (2) priority inversion. (Aside: I don't know why more people don't do this.)

   (a) When you schedule a thread, record the start time and when it blocks, record the end time. Check the difference and give an error if it's "too long." Alternatively, set a watchdog timer and "feed" it every time you schedule a thread. If it fires, the thread ran "too long".

   (b) For each lock (or blocking resource), record the thread priority that holds it, if higher priority thread waits on the lock/resource, give an error.

2. You build the network system in the livelock paper on top of MESA. There are two threads, both have the same priority. One thread runs the code for `screend` (translated into MESA). The other thread calls `process_pkts` in the following `Network` monitor:

```
// network hardware notifies this condition variable
// using the "well-known" wakeup waiting switch
condition_var has_packet;
ENTRY get_packet() -> packet BEGIN
    // if network interface has no packet, wait.
    while(nic.empty())
        wait(nic.has_packet);
    // pull packet off the nic and consume it.
    return nic.get_packet();
END
EXTERN process_pkts() BEGIN
   while(1) {
      // <forward> puts the packet on the
      // appropriate queue.
      forward(get_packet());
END
```

What two problems that the livelock paper discusses will this code have? How would you fix the code using MESA-appropriate versions of their solutions?
First problem, the `process_pkts` *code does polling without a quota so if packets come in fast enough it will run and never give screend a chance to run. Add a quota to the loop, when exceeded* `yield()` *which will run the other thread. (Answer should probably have some description of what to do when quota reached.)*

Second problem: the implementation never disables the nic if the `screend` *queue fills up. To do so the packet processing thread (in the given code) can check the* `screend` *queue and if full, sleep for 1ms or so (as in livelock.)*

3. Assume: `volatile` works as discussed in class: it guarantees the compiler will will not reorder `volatile` accesses with respect to each other, nor will it insert or eliminate `volatile` references.

   Assume as discussed in class that we have a simple compiler "memory barrier" that prevents the compiler from moving loads or stores across the barrier and forces all live registers to be written to memory before it and loaded from memory after.

   From the buggy code examples in the "threads can't be implemented as a library" paper: (1) explain one the programmer could prevent using either `volatile` or a barrier, and (2) explain one that neither `volatile` nor barriers could be used to eliminate.
   Could prevent both the speculation bug in 4.1 (since it would prevent speculation) and the speculation bug in 4.3 (since the compiler could not insert extra loads or stores to the protected data). Neither can eliminate the modification of "variables not mentioned in the source" or the problem of non-atomic stores/loads.

4. Eraser+Mesa: the eraser authors compare themselves to Mesa in the related work. Is their comparison completely accurate? Justify your answer.
   The wording is a bit confusing, but one reading is that they claim that MESA montiors (not just Hoare monitors) do not support "dynamically allocated shared variables." This is, of course, false since MESA has monitored records.

5. Eraser: the interrupt discussion in the Eraser paper is arguably a bit dubious. Assume the checked code runs on a uniprocessor and has a routine `disable()` to disable interrupts and `enable()` to enable them, but no notion of priority levels. Explain how to adapt eraser to catch bugs where non-interrupt code does not protect data shared with interrupt code.
   Goal: If non-interrupt code reads/writes data written by the interrupt handler without disabling interrupts you want to flag an error. How: It's a bit logically awkward but you could use annotations to have eraser believe disable() and enable() acquire/release a global interrupt lock, as does the interrupt handler. If this "lockset" goes to empthy flag the data. This isn't entirely direct but does not require modifying eraser.

6. Observation games: you see the following routine:

```
int foo(void) {
    int x,y=3;
    memcpy(&x,&y,sizeof x);
    return x;
}
```

   You compile `foo` with an optimizing compiler that understands the C standard library. What is the shortest equivalant code the compiler could generate for `foo`? Explain your reasoning.
   as discussed in class: if the compiler understands `memcpy` *it can determine that its invocation in this code is equivalant to the assignment* `x=y`*. Since the compiler knows* `y==3`*, then* `x==3`*, thus the entire routine reduces to the assembly code equivalant* `return 3;`*. Note that this isn't theoretical: the current (somewhat old) version of gcc on my laptop does this — I'd suggest trying with clang or gcc on yours!*

7. For "Knot-A" and "Knot-C" in Figure 3 (Section 5) in the "Why events are a bad idea." Use ideas from the livelock paper to (at least partly) explain the performance difference of these curves.
   Knot-A: favors accept, so will keep accepting work even when it cannot keep up with existing work (so is a push architecture). Thus, at a high enough rate, the throughput goes down dramatically (possibly down to 0). Knot-C: is similar to livelock solution in that it is "pull" architecture that goes an pulls more work when it has finished its existing

work. This means it doesn't get stuck in earlier pipeline stages given a high input rate. Nor will it discard packets that it has spent cycles working on. While the knot application has similarities to the livelock approach, we don't know about the underlying OS, which could have the same vulnerabilities. The paper speculates that interrupt overhead leads to the degredation. May well not account for fairness, but we can't see from the graph afaik.

8. You are running an identical, multiprogrammed workload in two virtual machines, A and B, on two separate physical machines. The following events transpire:

   (a) On the first physical machine, ESX enters the *hard* state and forcibly reclaims a host physical page from virtual machine A, writing the page contents to disk.

   (b) On the second physical machine, B's kernel evicts a page from a user process.

   Would A's expected throughput degrade more than B's, vice versa, or neither? Justify your answer.
   A's expected performance would degrade more.

   From ESX's point of view the guest OS is largely a black box. ESX has no way to "switch threads" in the guest. Thus, if a guestOS faults on a PPN that is paged out, the guestOS's whole virtual CPU is blocked. In contrast, a guestOS can easily switch to another process if one process blocks on a paged-out PPN.

   There were some answers that talked about blocking the guestOS on eviction, but that doesn't appear to be necessary — ESX can page out asynchronously.

9. Superpages: Assume we run the all the benchmarks in Table 1 concurrently (all at the same time), rather than sequentially (one-after-another) on the exact Alpha machine used in the paper. What broad changes would you expect for all the columns, especially the 4MB column?
   We should see a shift leftwards for the superpage usage columns—there should be many more base pages in use, and fewer large pages (maybe

even no 4mb pages). The runtimes will get slower (probably significantly), so the speedups will be negligible since many reservations would fail. We don't expect things to get much worse than baseline (running concurrently on an unmodified system), since even with all the promotion overhead in the adversarial experiments there isn't much degradation. We expect more TLB misses due to the smaller pages, and therefore a lower miss reduction percentage.

10. In a parallel universe, NaCl is wildly successful and x86 adds a NaCl jump instruction that performs both the `and` and indirect jump in a single instruction. What problem does this get rid of? Explain how you would change the verifier (figure 3) to make it as simple as possible and justify your answer.
    If nacl-jump is a single instruction, you don't have to worry about evil code jumping over the `and` *instruction used to 32-byte align the target. Since its no longer a 2-instruction sequence, we don't have to do the special check for the jump. As a result the if-statement that checks the nacl jump goes away, and the verifier simply always adds* `IP` *to* `JumpTargets` *(versus eliding indirect jumps.)*

11. The nacl group does the following optimization: if there is only a single call to a given routine `foo`, they replace the original call instruction to `foo` with a simple direct jump (that does not write the return address register), and `foo`'s return instruction (`ret`) with a direct jump back. Assume they do this transformation correctly: what speed and space improvement could this achieve?
    A direct jump is faster than the nacl-jump (both b/c less instructions and b/c you don't get a possible misprediction (you don't need this point)). Also, as they state: since indirect jumps can only jump to the start of a block, this forces any call instruction to be the last instruction in the block (otherwise the indirect jump done by return couldn't jump to the instruction after the call). Thus this constraint causes internal block fragmentation — which goes away without it.

Problem 12: superpages (10 points) Assuming Carl is willing to by default allocate machine memory in 4MB contiguous, superpage-aligned chunks: How could ESX be modified to transparently support guestOSes that implement superpages? How would page sharing complicate this approach?

Recall: The ESX paper talks about how to remap a GuestOS's virtual memory by (partially) translating the guest's page tables to shadow page tables controlled by ESX. You may need to propagate superpage information from the guestOS page table to the shadow page tables.
If we have 4mb-aligned machine pages that correspond to 4mb physical pages, the superpage hack should just work. Each 4mb superpage chunk that the guestOS uses would correspond to a correctly-aligned 4mb machine page that can be used interchangeably.

As ESX constructs shadow page tables, it can simply propogate whether the guestOS entry was a superpage or not, and (again) should should be able to simply relabel the PPN as its corresponding MPN without any problem since alignment is preserved.

With that said, ESX page sharing works at the level of 8k pages (base pages) and if you share pages these can split up large super pages. So the implementation would have to decide whether it wanted sharing more or superpages more or share at the level of superpages (maybe a good idea anyway! if one page is sharable, good chance the ones adjacent are as well).

Problem 13: Bugs and observations(10 points)

The Eraser paper states that the bug in section 4.2 is a serious race because of the lack of memory barriers. Let's say we go wild and throw memory `barrier()` calls around the statements guarded by the if-condition on the true path:

```
Combine::XorFPTag::FPVal() {
    if(!this->validFP) {
        barrier();
        NamesFP(fps, bv, this->fp, imap);
        this->validFP = true;
        barrier();
    }
    return this->fp;
}
```

Assume:

1. The compiler does not move code in either direction across `barrier()` calls and these calls work as expected by the Eraser paper.

2. That we simplify even further by only running the code on a uniprocessor.

3. That the compiler can see the implementation of `NamesFP`.

Two part question:

1. What are two ways variants of the problem could still arise because of transformations by an aggressively optimizing compiler? (Hint: for one problem, you may have to assume that the compiler does not consider the implicit "else" (false) path of the if-statement as having a memory barrier.)
   First, if we assume the compiler sees the implementation of `NamesFP` *(e.g., its an inline function). The compiler can potentially reorder setting* `validFP` *to true with some of the statements that initialize* `fp`.

   Second, since `fp` *and* `validFP` *are known to be disjoint fields, the C++ compiler could potentially hoist the read of* `this->fp` *before the check. This wouldn't break sequential code, but can break our threaded code.*

2. If you can fix this code by adding barriers, do so. If not argue why you can't.
   easiest fix is to put a barrier before the `return` *and between the* `NamesFP` *and* `validFP=true` *statements (to preserve ordering).*