# CS 240 Midterm Examination

Fall Quarter 2025

**You have 80 minutes** for this examination; the number of points for each question indicates roughly how many minutes you should spend on that question. Make sure you print your name, SUID #, and sign the Honor Code below.

**This is an open-book exam**. You may bring any amount of printed material; all other sources of information, including laptops, cell phones, etc. are prohibited. No electronic devices are allowed.

Write all of your answers directly on the paper in the spaces provided after each question.

**NOTE: We will deduct points if a correct answer includes incorrect or irrelevant information. (i.e., don't write in everything you know in hopes of saying the correct buzzword.)**

I acknowledge and accept the Stanford University Honor Code. I have neither given nor received aid in answering the questions on this examination.

_____

(Signature)

**Sample Answers**

_____

(Print your name, legibly!)

_____@stanford.edu

(Stanford email account for grading database key)

**Only the front side of the exam pages will be scanned. Do not write answers on the back of the pages.**

## Problem #1 (14 points)

Your CS140E partner has decided to implement monitors for the synchronization needs of your team's kernel. Their design has a couple of differences from what was described in the Mesa paper. For each of the design changes below, state whether the change would work or not. If it works, how would it compare with the approach used in Mesa? If it doesn't work, explain why. Assume Mesa-style monitors and condition variable semantics.

(a) Rather than implementing a `Broadcast` method on the conditional variable, they extend the conditional variable with a method that returns the number of threads waiting and the broadcast functionality was to be implemented by calling `Notify` that number of times.

**ANSWER:**
It would work. Replacing a Broadcast call with a call that returns the count of waiting threads, followed by a loop that uses Notify to wake them up, appears to open up a potential race condition between the count being read and the wait list being modified. The fact that any thread calling Broadcast must hold the monitor lock means this race cannot happen. The monitor lock means that no other Wait or Notify can alter the condition variable's wait list membership between the read of the wait count and the Notify loop. The Notify loop would wake all waiters on the list, and no other thread could join the list during the operation.

(b) Rather than implementing both the Mesa `Notify` and `Naked Notify` functionality, they proposed implementing only the `Naked Notify` mechanism and having the regular `Notify` built on top of it. In their design, there was to be a single underlying notification implementation that works for both the ordinary `Notify` and `Naked Notify` cases.

**ANSWER:**
ANSWER:
It would work with a possible performance hit from spurious wake-ups. The Naked Notify was done in Mesa to allow a device to do a Notify on a condition variable without holding the monitor lock. The lack of a monitor lock creates a race condition in which the device-handling thread sees that the device is not requesting attention, so it waits for the interrupt. If the device interrupts between this attention check and the waiting, the device interrupt Notify won't find the thread waiting, followed by the thread, then waiting after missing the interrupt. The Naked Notify handles this situation by having the Naked Notify set a binary semaphore if there is no one waiting, so that the next Wait will do a DOWN on the semaphore and not wait. The device thread that hits this race will not wait, re-examine the device, and handle the interrupt.

This race cannot happen on regular Notify because the monitor lock will always be held. Having such a semaphore causes the following thread that calls wait after a notify not to wait, leading to spurious wakeups on the conditional variable. Since Mesa-style monitors must always check after being woken up, these extra wakeups won't break anything, but can cause unnecessary additional work.

## Problem #2 (7 points)

The Mesa system pushed hard on using threads rather than events. Ironically, some of their optimizations for threads, such as fast `FORK`/`JOIN`/`Detach`, eliminated the need for the data copying that would have been required as part of stack ripping if events were implemented on the same machine. Describe what this was.

**ANSWER**:
Mesa's threads allocate procedure activation records from the heap rather than the stack used the papers that mention stack ripping. Stack ripping is the copying of the data from the stack since the original copy is lost when the activation record is popped from the stack. Allocating the records heap eliminates this requirement for copying since the record isn't forced to be deallocated. Hence, stack ripping isn't needed.

## Problem #3 (7 points)

The *Eraser* system worked on programs using the `pthreads` library. That library was also used in the parallel-processing community to coordinate threads working together on a single problem. As in the paper, such programs typically start with one thread; when execution reaches a parallel section, multiple threads run concurrently, each operating on its own portion of the data. The original thread then waits for all the others to finish before continuing. Execution repeatedly alternates between a single thread working on the data and multiple coordinated threads working on the data.

Explain what *Eraser* would report if it were run on such a program. Would it identify real race conditions, produce incorrect warnings, or fail to run at all? Be specific about why.

**ANSWER**:
Eraser assumes that locks are used to create critical sections, which is not the case in this programming style. It will unlikely do anything useful. An access pattern with many threads operating on different parts of a data structure shouldn't generate any messages, but the following sequential part and future parallel sections will likely cause Eraser to generate warnings about missing locks due to access to the same data items from different threads.

## Problem #4 (7 points)

In the Adams and Agesen paper, they compared their software virtual machine monitor — implemented using a dynamic binary translator—with a hardware virtual machine monitor on several benchmarks. High-performance Java virtual machine implementations typically use just-in-time (JIT) code generation, which usually performs poorly under binary translation. Despite this, the SPECjbb Java benchmark results showed that the software virtual machine monitor avoided these performance problems. How was this accomplished?

**ANSWER**:
The VMware VMM uses direct execution for user-level (ring3) code. The SPECjbb runs JITed Java code in user mode, so any problems with JITs and the binary translator aren't seen.

## Problem #5 (14 points)

Both modern operating systems and virtual machine monitors emphasize **isolation**. Modern operating systems advertise that different processes — possibly owned by different users — are isolated from one another and from the operating system kernel. Similarly, virtual machine monitors isolate the multiple virtual machines running on a single host.

Consider the virtual machine monitor described in the *Waldspurger ESX* paper, and answer the following questions:

(a) Can different processes owned by different users running in the same virtual machine end up sharing a page in machine memory even though the isolation properties of the guest operating system would not have set up such sharing? If not, explain why. If so, explain what could cause such an isolation violation

**ANSWER:**
With its transparent content-based page sharing mechanism, ESX can deduplicate a guest's memory content. If two processes in the same guest happen to have identical pages, they can end up using the same physical frame. This is the case for VM 1 illustrated in Figure 3. This mechanism is only used when pages have the same content, therefore it does not provide any additional information to either process and does not violate isolation.

(b) Can processes running in different virtual machines access the same page in machine memory, given the ESX isolation policy? If not, explain why. If so, explain how.

**ANSWER:**
The answer is the same here, and corresponds to the white pages in VM 1, VM 2 and VM 3 all pointing to the same physical frame in Figure 3.

## Problem #6 (14 points)

x86 operating systems such as Linux and Windows use superpages (4 MB or 2 MB) to map the kernel into memory. Answer the following questions about ESX's memory virtualization. ESX could have used superpages in its shadow page tables for these mappings but did not.

(a) Describe the disadvantage of not using superpages in the shadow page tables.

**ANSWER:**
The hypervisor has to maintain the shadow page tables, and each entry is 4k, so accessing each of them triggers some TLB pollution. Putting the shadow tables in superpages (for instance by placing a page directory's page tables right after it in memory) would make shadow page maintenance more efficient (and would make it reduce the guest's performance less)

(b) List examples of the benefits of using 4 KB page mappings in the shadow page tables. Include at least one example from each of the two VMware papers.

**ANSWER:**
Superpages would introduce fragmentation for the hypervisor, and page alignment in the guest's physical memory would have to be aligned in machine memory as well. This means that the tracing mechanism in the Adams and Agesen paper would have to be made much more complex, even though it is claimed that its performance is critical. Superpages also prevent fine grained page fault handling, which would make it almost impossible to implement Waldspurger's memory sharing mechanism. Further, if superpages could still somehow be shared, there would still be much fewer instances of actual possible sharing, since the occasions in which two much larger regions have the same content would be much less common, so each VM would occupy more memory.

## Problem #7 (7 points)

In virtualization theory, virtual machines are allowed to have timing differences from native machines. In practice, VMware found this mostly to be true. One counterexample was in booting a version of DOS that did an OUT instruction to some device on the motherboard and spun in a loop, doing IN instructions until the IN returned some value. VMware's much faster IN instruction meant the loop exited with a blue screen error message saying the hardware had failed. About the same time, AMD released a new CPU that made certain old x86 instructions run in fewer cycles and managed to hit a similar boot failure.

Explain why it was an easy software fix for VMware and a significant problem for AMD.

**ANSWER:**
~~VMware products are implemented in software.~~ VMware's products dynamically rewrite the guest kernel, therefore by patching the VMM it is possible to introduce patches that will emulate lengthened IN instructions. Furthermore, if an issue is known to arise only on a specific operating system and for a specific device, VMware can let users specify their operating system so that the VMM can exceptionally pace its IN instructions to that device. AMD, on the other hand, cannot easily patch their chips, and even if they did, making such an exception in hardware would be a poor design choice.

## Problem #8 (7 points)

In the *Receiver Livelock* paper, the authors needed to monitor the kernel queue feeding the `screend` process to avoid livelock. Recall that input processing was inhibited when the screening queue was 75% full and re-enabled when the queue dropped to 25% full.

Assume the architecture of screend is updated to run as a **pipeline** of multiple processes connected by standard Unix pipes. The first process receives each packet, which is then forwarded through the subsequent processes, with the last one sending the packet back out. Recall that Unix pipes are flow-controlled, so a write on a pipe will block until there is sufficient buffer space for the receiving process to accept the packet.

Would we now need to inhibit input based on the state of **each** pipe in the pipeline, or only on the **kernel-to-first-process** queue? Explain your answer.

**ANSWER:**
If we represent the situation described in the question as a diagram in the style of Figure 6-2 from the Livelock paper, we would get the same diagram!
In this diagram, packets can only be lost at the places where they could already be lost when screend was the only process involved. The new pipes can also experience congestion but they cannot drop messages, therefore the pressure on these pipes will backpropagate all the way to the first packet queue which will drop packets. Therefore, feedback from the first process in the pipeline reflects pressure in the whole pipeline, and no additional source of feedback is necessary.