CS 240 Midterm Examination

Fall Quarter 2026

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, email, 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)

_____
(Print your name, legibly!)

_____@stanford.edu
(Stanford email account for grading database key)

Problem #1 (16 points)

Consider the following Mesa-style monitor that implements a single-element mailbox. The code uses C-like syntax for convenience.

```
typedef struct {
    int value;
} Buffer;

MONITOR Mailbox {
    Buffer *slot = NULL;
    CONDITION notEmpty, notFull;

    ENTRY void send(Buffer *p) {
        while (slot != NULL)
            WAIT(notFull);

        slot = p;
        NOTIFY(notEmpty);
    }

    ENTRY Buffer *receive() {
        while (slot == NULL)
            WAIT(notEmpty);

        Buffer *p = slot;
        slot = NULL;
        NOTIFY(notFull);
        return p;
    }
}
```

One producer thread repeatedly allocates, initializes, and sends buffers:

```
    while (1) {
        Buffer *p = malloc(sizeof(Buffer));
        p->value = 42;
        Mailbox.send(p);
    }
```

Two consumer threads repeatedly receive, modify, and free buffers:

```
    while (1) {
        Buffer *p = Mailbox.receive();
        p->value++;
        free(p);
    }
```

… problem #1 continued from previous page

Assume that Eraser recognizes the Mesa monitor's locking operations, threads, and the `malloc` and `free` operations. The producer does not access a buffer after sending it, and each buffer is received by exactly one consumer.

(a) (4 points) Will Eraser report a race on the mailbox's `slot` variable? Explain your answer.

ANSWER:
No. All accesses to `slot` occur while holding the mailbox's monitor lock. WAIT releases the lock while sleeping but reacquires it before execution resumes. Therefore, the monitor lock remains in Eraser's candidate lockset for slot, and Eraser reports no race.

(b) (4 points) Suppose we swap the last two statements of `send`, so that `NOTIFY(notEmpty)` executes immediately before `slot = p`. Would the mailbox still operate correctly? Would Eraser report any additional race warnings? Explain both answers.

ANSWER:
The mailbox remains correct, with no additional Eraser warnings. Mesa NOTIFY does not transfer the monitor lock to the awakened consumer. The producer still holds the lock while executing slot = p, so a consumer cannot inspect slot before that assignment. Consumers recheck the condition in a while loop after acquiring the lock. The reordering changes neither the locks held during accesses nor the buffer-ownership protocol.

(c) (8 points) Will Eraser report a race on the `value` field of a buffer? If so, does this indicate an actual race in the program? Explain your answer.

ANSWER:
Yes, Eraser reports a warning, but it is a false positive. The producer's initialization puts value in the Exclusive state. The consumer's subsequent read and write move it into Shared-Modified. Because the consumer holds no lock during the increment, the candidate lockset is empty.

There is no actual race: the monitor synchronizes publication of the initialized buffer, the producer never accesses it afterward, and exactly one consumer receives it. This safely transfers exclusive ownership. Eraser checks for consistent locking but does not infer this ownership transfer. Recognizing malloc and free prevents confusion between different allocations that reuse an address.

Problem #2 (14 points)

VMware ESX supports two mechanisms for reclaiming machine memory from a virtual machine: ballooning and paging directly to an ESX-managed swap file.

Consider a VM with a single virtual CPU running a guest OS with two runnable processes, A and B.

(a) (4 points) Explain how ESX reclaims machine memory using each mechanism. In particular, who decides which pages to reclaim?

ANSWER:
Ballooning: ESX asks a guest balloon driver to allocate and pin guest-physical pages. The guest OS supplies free pages or chooses pages to reclaim using its own policy. The driver identifies its allocated pages to ESX, which reclaims their backing machine frames. ESX chooses how much memory to reclaim; the guest chooses which existing uses to displace.
Direct paging: ESX itself selects guest-physical pages, saves their contents in its swap file, and reclaims their machine frames without involving the guest OS.

(b) (6 points) Suppose that under either mechanism, a page belonging to process A is evicted to disk. Process A subsequently accesses that page. Compare what happens under the two mechanisms. Which software handles the page fault? Can process B execute while the page is being read from disk?

ANSWER:
Ballooning: The guest OS evicted A's page and knows it is absent. It handles A's page fault, starts the disk read, and blocks A. The guest scheduler can run B while waiting, assuming B's required pages are resident.
Direct ESX paging: The guest still believes the page is resident; its machine-memory backing is missing. ESX handles the fault and stalls the virtual CPU while reading the page. The guest scheduler cannot run to switch from A to B. With only one virtual CPU, B cannot execute during this wait, although ESX can run another VM.

(c) (4 points) Suppose the guest OS uses much of its memory for a file-system cache containing clean pages. Explain why ballooning might reclaim memory without writing anything to disk, whereas direct ESX paging might require disk writes.

ANSWER:
The guest OS knows that clean file-cache pages have up-to-date copies on disk. Under balloon pressure, it can discard those cached copies and give the freed frames to the balloon without writing anything.

ESX generally lacks this knowledge of the guest's file cache. Direct paging must preserve the contents of the guest-physical pages it evicts, so ESX may write them to swap even though the guest could have discarded them.

Problem #3 (14 points)

The Navarro et al. superpages paper uses a reservation-based allocation scheme to preserve physical memory contiguity, allowing pages to be promoted into superpages without copying their contents.

The Alpha processor used in the paper has a software-managed TLB, with TLB misses handled by PALcode. Suppose you could modify these handlers to:

- Record which virtual addresses cause TLB misses.
- Maintain TLB-miss counts for different virtual-address regions.
- Make this information available to the operating system.

You cannot change the TLB hardware.

(a) (10 points) How could these capabilities change the design of a superpage management system? Describe an alternative to the paper's reservation-based approach. Explain how your design would decide when to create a superpage and what work would be required to create it.

ANSWER:
Allocate ordinary base pages initially, then use the TLB-miss counts to identify virtual regions that would benefit from superpages. Track recent misses per address space and aligned region. Promote when the expected future savings from fewer TLB misses justify the promotion cost; frequent misses across many pages in one region provide evidence of such a benefit.

Because the base pages may be scattered, obtain a contiguous, properly aligned physical region and copy the pages into it in virtual-address order. Update the mappings and page-size information, invalidate stale TLB entries, and release the old frames. Coordinate accesses during relocation so that writes are not lost. The pages must have compatible protection attributes; a simple design can restrict promotion to fully populated regions.

If contiguous space is unavailable, compact memory or defer promotion. Unlike reservation-based allocation, this approach creates contiguity after observing demand and therefore pays copying and monitoring costs.

… problem #3 continued…

(b) (4 points) Your colleague suggests that, since the TLB is software-managed, you could simply modify the PALcode handlers to support superpages even when their constituent base pages are scattered throughout physical memory.

Would this work? Explain why or why not.

ANSWER:
No. Software controls how the TLB is filled, but the hardware determines what each entry can represent. A superpage entry maps a contiguous virtual region to a contiguous, aligned physical region using a physical base plus an offset. It cannot encode an arbitrary list of scattered frames.

PALcode runs on TLB misses, not on hits, so changing the handler cannot change that translation rule. Separate base-page entries can map scattered pages, but they do not provide the TLB coverage of one superpage entry.

Problem #4 (14 points)

You are implementing a web server that handles two types of requests:

- **Authentication requests:** Verify passwords using a cryptographic hashing library. To make password guessing expensive, the library deliberately consumes 500 milliseconds of CPU time per authentication.
- **Simple requests:** Require less than 1 millisecond of CPU time.

The cryptographic library is synchronous: once called, it computes continuously until it returns. It never yields or blocks waiting for I/O, and you cannot modify it.

You are considering three server implementations:

1. **Events:** A single-threaded event loop processes requests using event handlers that run to completion.
2. **Cooperative threads:** Each request runs in a separate user-level thread. These threads share one kernel thread and switch only when the running thread voluntarily yields or blocks through the threading library.
3. **Preemptive threads:** Each request runs in a separate kernel thread, scheduled preemptively by the operating system.

Assume the server runs on a single CPU and the preemptive scheduler uses a 10-millisecond time slice with fair scheduling.

(a) (6 points) An authentication request begins executing. Ten milliseconds later, a simple request arrives. For each of the three server implementations, explain whether the simple request can complete before the authentication request finishes. Explain why.

ANSWER:
Events: No. The authentication handler runs to completion, preventing the event loop from handling the simple request for approximately the remaining 490 ms.

Cooperative threads: No. The hashing library neither yields nor blocks through the threading library. The authentication thread therefore retains the only kernel thread, and the simple request's user-level thread cannot run.

Preemptive threads: Yes. The OS can preempt authentication at a time-slice boundary and schedule the simple request's kernel thread. Its less than 1 ms of work fits within one 10 ms slice, so it can finish well before authentication.

… question 4 continued …

(b) (4 points) Consider another type of request that performs the following operations:

```
record = read_disk(user_id);
preferences = read_disk(record.preferences_id);
send_response(record, preferences);
```

Assume the disk operations may take a long time. Asynchronous disk I/O is available, and the cooperative threading library can suspend only the calling user-level thread while its I/O is pending, allowing other user-level threads to run. The server should process other requests while waiting for disk I/O.

Explain how this request would be implemented differently using events versus cooperative threads. In particular, how does each approach preserve the request's execution state while waiting for disk I/O?

ANSWER:
Events: Split the request into callbacks: start the record read and return; on completion, start the preferences read; on its completion, send the response. Preserve information such as the client connection and record in a request-state object that survives between callbacks.

Cooperative threads: Keep the sequential code. Each read starts asynchronous I/O and suspends only the calling user-level thread. Its stack and saved execution context preserve local variables and where to resume. Other threads run while the I/O is pending.

In either approach, the second read must wait for the first because it needs record.preferences_id.

(c) (4 points) You want to modify the event-driven implementation so that simple requests can complete promptly even while authentication requests are being processed. You cannot modify the cryptographic library, and the server still runs on a single CPU.

Describe a modification that achieves this goal. Would your modification increase the maximum throughput of authentication requests? Explain.

ANSWER:
Offload hashing to a separate kernel thread or worker process. The event loop queues authentication work and handles other requests; the worker notifies it asynchronously when finished. The OS can preempt the worker so the event loop remains responsive even on one CPU. The event loop must not wait synchronously for completion.

This improves latency, not maximum authentication throughput. Each authentication still consumes 0.5 CPU-seconds, limiting one CPU to at most **2 authentications per second** before other work and overhead. Additional workers cannot increase that CPU capacity.

Problem #5 (11 points)

The *Eliminating Receive Livelock in an Interrupt-driven Kernel* paper describes an approach to preventing receive livelock.

Consider an alternative design. Instead of using network receive interrupts, the OS uses a periodic timer interrupt to wake up a network polling thread. When scheduled, this thread processes incoming packets until the network interface has no more packets waiting, at which point it blocks until the next timer interrupt.

Assume the polling thread is preemptively scheduled at the same priority as ordinary application threads, with fair CPU scheduling.

(a) (5 points) Would this design prevent receive processing from starving runnable application threads of CPU time, even under sustained overload? Explain why or why not.

ANSWER:
Yes. Under overload, the polling thread may never empty the interface and therefore may never voluntarily block. However, it remains preemptible and has the same priority as application threads. Fair scheduling gives those applications CPU time even while the poller remains runnable. Packet processing therefore cannot monopolize the CPU as high-priority receive interrupts can. Queues may still overflow and packets may be dropped; avoiding CPU starvation does not eliminate overload.

(b) (6 points) How would the choice of timer interval affect the performance of this design? Discuss both lightly loaded and heavily loaded conditions.

ANSWER:
Light load: A shorter interval reduces the wait for a sleeping poller to wake, but increases timer interrupts, wakeups, and empty polls. A longer interval reduces overhead but increases latency. With interval T and uniformly timed arrivals, the average wait for the next tick is T/2, plus scheduling delay.

Heavy load: If the poller sometimes sleeps, longer intervals can produce larger batches and amortize wakeup costs, but also increase queueing and the risk of receive-buffer overflow.

Under sustained overload, the poller may never sleep. It resumes whenever scheduled, without waiting for another polling timer tick. The interval then has little effect on steady-state processing, apart from timer overhead. CPU allocation is controlled by the scheduler; the polling interval is not a CPU budget.

Problem #6 (11 points)

In the Adams and Agesen paper, the hardware VMM was often slower than the software VMM. Suppose Intel proposes two improvements to the hardware virtualization support described in the paper:

A. Faster VM transitions: Reduce the overhead of a VM exit followed by a VMRUN to resume guest execution to that of an ordinary x86 trap and return. Assume the time spent executing VMM code is unchanged.

B. Nested paging: Add hardware support for translating guest virtual addresses to machine addresses without requiring the VMM to maintain shadow page tables.

(a) (9 points) Consider the following workloads from the paper:

1. **SPECint2000:** Primarily user-level computation.
2. **ptemod:** Repeatedly modifies guest page-table entries.
3. **in:** Repeatedly executes an I/O port input instruction.

For each workload, explain which improvement(s), A or B, you would expect to significantly reduce its total execution time and why. Evaluate A and B independently, each relative to the original hardware VMM described in the paper. You may choose A, B, both, or neither. You do not need to predict which improvement provides the larger speedup.

ANSWER:
SPECint2000: Neither. Most time is spent in user computation that already executes directly. VM transitions and shadow-page-table maintenance account for relatively little execution time, so neither improvement targets a dominant cost.

ptemod: Both. A makes the frequent exits and resumptions caused by guest page-table writes cheaper, although VMM processing remains unchanged. B eliminates the need to intercept those writes merely to keep shadow page tables consistent, removing that source of exits and maintenance work.

in: A only. Each port-input operation requires VMM intervention, so cheaper transitions reduce its repeated overhead. Nested paging changes memory translation and does not remove the need to handle the I/O instruction.

(b) (2 points) Nested paging uses an additional page table to translate guest-physical addresses to machine-physical addresses. What potential performance disadvantage does this introduce compared with shadow page tables?

ANSWER:
Nested paging can make TLB misses more expensive. A shadow page table directly maps guest virtual addresses to machine addresses. With nested paging, accessing guest page-table entries may itself require guest-physical-to-machine translation, as does the final data address. The combined walk can therefore require many more memory accesses. TLB hits can use a cached combined translation, so this extra walking is not required on every memory access.

This page was intentionally left blank