CS 240 Final Examination Solutions

Fall Quarter 2025

You have 180 minutes (3 hours) 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)

Solution

_____
(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 (18 points)

The Sun NFS paper included a stated goal of "UNIX Semantics Maintained on Client". This turned out to be quite a challenge since UNIX allowed some weird file system semantics.

Assume you have an NFS cluster with a single NFS file server (`Server`) exporting a file system to two clients: `ClientA` and `ClientB`. Each client machine, as well as the server, has a process that has opened the same large file on the shared file system and is performing constant reads from it.

For each of the scenarios below, state whether any of the processes running on `ClientA`, `ClientB`, or the `Server` encounters an error or otherwise misbehaves. Specify which machines experience problems and what those problems are. Then answer the same scenario assuming the NFS server quickly reboots immediately after the operation.

A. A program running on `ClientA` renames the file to a different name in the same directory.

ANSWER:
No processes encounter any problems. The programs on all three machines continue to read the file successfully. Each program holds an open file descriptor that refers to the file's vnode/inode, which is independent of the directory entry name. Renaming the file only changes the directory mapping and does not affect existing open file descriptors. On the NFS clients, operations are issued using a file handle that identifies the file by inode (and filesystem), not by pathname, so the server continues to service the read requests normally.

After Server quickly reboots:

ANSWER:
The program running on the server will terminate and stop reading the file. After reboot, it would need to reopen the file under its new name in order to continue. The programs running on ClientA and ClientB will continue to read the file successfully: their NFS requests use file handles that identify the file by inode (and filesystem), not by name, and the stateless nature of the NFS server allows it to resume servicing these requests after reboot.

… Problem #1 continued from previous page

B. A program running on `ClientB` removes the file.

ANSWER:
No processes encounter any problems. The programs running on all three machines continue to read the file successfully. This scenario exercises NFS's handling of deleting an open file. When a client removes a file that is still open, the NFS client typically implements the delete as a rename to a hidden temporary file so that the file's inode remains reachable until the last open reference is closed. As in part A, each program holds an open file descriptor referring to the file's inode, so changing the directory entry does not affect ongoing reads. NFS requests continue to be serviced using file handles that identify the file by inode, not by name..

Server quickly reboots:

ANSWER:
The program running on the server will terminate and stop reading the file; after reboot it would need to reopen the file (if it still exists) to continue. The programs running on ClientA and ClientB will continue to read the file successfully: their NFS requests use file handles that refer to the inode, and the stateless design of the NFS server allows it to resume servicing these requests after reboot.

… Problem #1 continued from previous page

C. A program running on `Server` renames the file.

ANSWER:
No processes encounter any problems. The programs running on all three machines continue to read the file successfully. Renaming the file only changes the directory entry; it does not affect the inode or any existing open file descriptors. As described in the NFS paper, open files remain accessible via their inode even if their name changes. The NFS clients use file handles that identify the file by inode (and filesystem), so reads from ClientA and ClientB continue unaffected.

Server quickly reboots:

ANSWER:
The program running on the server will terminate and stop reading the file; after reboot, it would need to reopen the file under its new name to continue. The programs running on ClientA and ClientB will continue to read the file successfully. Although the server reboot causes all server-side state to be lost, the file itself still exists on disk under the same inode, and the stateless NFS server can resume servicing requests using the existing file handles held by the clients.

Problem #2 (12 points)

Assume we extend NFS with the lease mechanism described in the *Leases* paper. Would this leased NFS system be able to claim *primum non nocere* (Latin for “first, do no harm”) in the same sense as the *Superpages* paper?

Argue either **yes** (and explain why), or **no** by identifying workloads in which NFS with leases would both **improve** performance and **degrade** performance relative to standard NFS.

ANSWER:
No. NFS with leases cannot claim primum non nocere. While leases can improve performance for some workloads, they can also degrade performance for others relative to standard NFS.

Leases strengthen NFS’s consistency guarantees by allowing clients to cache data more aggressively without frequent revalidation. However, many workloads that run correctly and efficiently on existing NFS systems do not require these stronger guarantees. For such workloads, leases can introduce new overheads.

Improve performance:
Consider a workload in which multiple NFS clients repeatedly read files from a server. In standard NFS, clients must periodically perform validation (e.g., GETATTR calls) to ensure cached data is up to date. Leases eliminate much of this revalidation traffic by granting clients time-bounded guarantees, reducing RPC overhead and improving read performance.

Degrade performance:
Now consider a workload with multiple NFS clients writing files that is carefully structured to work correctly under standard NFS’s weaker consistency model. With leases, write operations may be delayed while the server revokes leases held by other clients caching the file. These revocation delays introduce additional latency and coordination overhead, reducing performance for workloads that did not require tighter consistency in the first place.

Problem #3 (12 points)

The F2FS file system differed from LFS in that it had multiple logs. The following table from the paper shows the 6 different logs.

Table 1: Separation of objects in multiple active segments.

Type	Temp.	Objects
Node	Hot	Direct node blocks for directories
	Warm	Direct node blocks for regular files
	Cold	Indirect node blocks
Data	Hot	Directory entry blocks
	Warm	Data blocks made by users
	Cold	Data blocks moved by cleaning; Cold data blocks specified by users; Multimedia file data

A. How did the F2FS designers decide which block *types* should be treated as hot, warm, or cold in this table? Explain your answer.

ANSWER:
The classification is based on the frequency at which blocks are predicted to be written to. This multi-headed where each log contains data with similar update frequencies makes garbage collection more efficient and reduces the write amplification factor.

B. Why would some types of node blocks be considered Hot or Warm while others are considered Cold? Explain your answer.

ANSWER:

- A direct node block for a directory is updated every time the directory entry it points to is modified, which is, in turn, very frequent because it happens whenever a file inside that directory is added/removed. This is classified “Hot” in the same table (for data blocks), so the direct ricochet effect of these events should also be “Hot”.
- A direct node block for a regular file is updated every time the data it points to is modified. This is classified “Warm” in the same table (for data blocks), so the direct ricochet effect of these events should also be “Warm”.
- Indirect node blocks are not updated when a data block it points to is modified, so it is modified very rarely (when a node block it points to is added or removed). That is thanks to NAT solving the “wandering tree” problem.

Problem #4 (10 points)

Explain what threading the log means and why this design made more sense for F2FS than for the original LFS.

ANSWER:
Threading the log means reusing free regions inside partially filled segments by logically linking (“threading”) the log through these holes, rather than reclaiming space only by cleaning entire segments and writing data sequentially into newly allocated ones. This allows the filesystem to avoid copying live data out of partially full segments simply to create contiguous free space.

LFS considered this approach but rejected it because, on disks with high seek costs, writing a log that jumps between scattered free regions would require frequent seeks, significantly reducing write bandwidth and defeating the benefits of sequential log writing.

F2FS, in contrast, targets solid-state storage where seek costs are negligible. As a result, threading the log through freed regions does not incur the same performance penalty, making this design both practical and beneficial for reducing write amplification on flash-based devices.

Problem #5 (10 points)

The LFS paper includes a microbenchmark that creates 10,000 1K files on a newly created file system. The storage was large enough that no cleaning was necessary.

ANSWER:
How would you expect F2FS performance on this microbenchmark to compare to LFS if both were running on an SSD?

Creating 10,000 small (1 KB) files is close to a best-case workload for LFS. The file data, inodes, and directory blocks can be accumulated in memory, packed densely into segments, and written sequentially to disk, allowing LFS to achieve near-peak write bandwidth with minimal overhead and no cleaning.

F2FS would also perform well on this workload, especially on an SSD, but it incurs additional overhead relative to LFS. In particular, F2FS must update its Node Address Table (NAT), Segment Information Table (SIT), and maintain multiple logs for different block types. These metadata updates introduce extra writes and coordination that LFS largely avoids in this scenario. Moreover, F2FS enforces more disciplined metadata updates to support efficient garbage collection and crash recovery, reducing its ability to batch updates as aggressively as LFS.

As a result, while both filesystems perform well, LFS would be expected to outperform F2FS on this specific microbenchmark.

Problem #6 (18 points)

Both Salus and vLLM focused on improving memory usage efficiency to increase concurrency. Despite this commonality, the papers highlighted different techniques and approaches. Answer the following questions about the two systems:

A. Salus was able to focus on **iterations**, while this wasn't useful for vLLM. Explain why vLLM could not make use of iteration boundaries.

ANSWER:
Salus focused on deep learning training workloads, which have a well-defined iteration structure: each iteration repeatedly allocates memory, performs computation, and then frees memory. These iteration boundaries give Salus natural points at which memory can be reclaimed, defragmented, and safely reallocated across jobs.

vLLM, in contrast, targets inference workloads, where requests arrive continuously and do not have a fixed iteration structure. Memory usage grows dynamically as sequence lengths increase, and there is no clear point at which memory associated with a request can be reclaimed until the request completes. As a result, vLLM cannot rely on iteration boundaries and instead focuses on keeping the KV cache resident and managing it efficiently as requests progress

B. Salus had a "**safety" condition** that guided their scheduler. Explain why vLLM didn't need a "safety" condition.

ANSWER:
Salus's scheduler needed a safety condition to decide which jobs could be co-located in the same memory lane. Training jobs have both persistent memory (model parameters) and large, ephemeral per-iteration allocations. The safety condition ensured that, at any iteration boundary, the sum of persistent memory and worst-case ephemeral allocations of all co-running jobs would fit within the lane's physical memory, preventing out-of-memory failures.

vLLM, in contrast, manages inference workloads whose dominant memory usage is the KV cache, which grows incrementally as sequences progress. vLLM does not need a Salus-style safety condition because it allocates KV cache memory in fixed-size blocks and only admits new requests when sufficient memory is available. Memory growth is controlled dynamically through block allocation rather than predicted ahead of time, eliminating the need for conservative worst-case guarantees across co-located jobs.

C. Salus featured **Auto Defragmentation.** Explain why vLLM didn't need Auto Defragmentation.

ANSWER:
Salus performs Auto Defragmentation at iteration boundaries. When an iteration completes, the ephemeral memory allocated by training jobs is no longer in use, allowing Salus to compact the remaining live allocations within a lane and eliminate fragmentation before the next iteration begins.

vLLM does not require Auto Defragmentation because inference workloads do not exhibit large, short-lived bursts of ephemeral allocations with clear quiescent points. Instead, vLLM's primary memory consumer is the KV cache, which grows monotonically during request execution. By allocating KV cache memory in fixed-size blocks and managing it via a paging-like indirection, vLLM avoids external fragmentation altogether and therefore has no need for compaction.

Problem #7 (10 points)

A state-of-the-art LLM Chatbot states that "vLLM implements zero-copy reallocation when sequence lengths grow". Explain what is being reallocated and what is not copied?

ANSWER:
When the sequence length grows, the KV cache needs to grow with it. This was previously done by either provisioning the KV cache with potentially too many blocks (leading to what the paper refers to as “internal fragmentation”), or copying the KV cache to a memory region with enough free space to fit the current and the next blocks. With vLLM, the reallocation operation allowing the expansion of the KV cache of a sequence is done without copying the KV blocks that had already been computed.

Problem #8 (18 points)

Let's say we want to run a VMM as a VM of another VMM. To avoid possible confusion, let us call L0 the software really running in the privileged execution mode. Let us call L1 the software virtualized by L0 (in a classical setting, this is what would be called the "guest kernel", except here it is actually a "guest hypervisor"). Let L2 be the software virtualized by L1 (in a sense, the real "guest kernel"). If L2 happened to be, in turn, a hypervisor, then a guest of L2 would be called L3, and so on. For the rest of this problem, we assume that all of the hypervisors at every level run the same code.

A. Say L2 runs an unprivileged application which invokes the `syscall` instruction. Describe how the syscall would get routed from the application to the L2 kernel, both for a hardware VMM and for a software VMM.

ANSWER:
For a hardware VMM:

- The `syscall` instruction makes the process go from L2's non-root ring 3 to L2's non-root ring 0

For a software VMM:

- The `syscall` instruction is run natively because it is triggered by a userspace program
- It traps to the only piece of software running in privileged mode, which is L0
- L0 will reflect it to L1 and execute it with its binary translator in unprivileged mode
- L1 will think that it comes directly from L2 and will therefore try to reflect it to L2 just like L0 reflected it to L1
- L1 tries to context switch to L2, but this requires privileged instructions which can only be run from L0 (the Adams and Agesen paper says "Complex operations like context switches call out to the runtime") so L0 starts running again
- L0 reflects L1's reflection to L2

… Problem #8 continued from previous page

B. If the application was running in Ln rather than L2, how many privilege transitions would happen as a function of n, for both a software VMM and a hardware VMM?

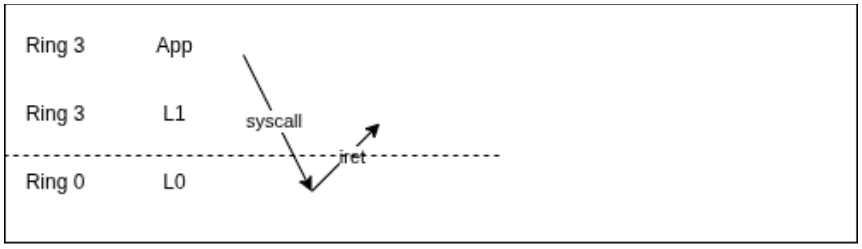
ANSWER:

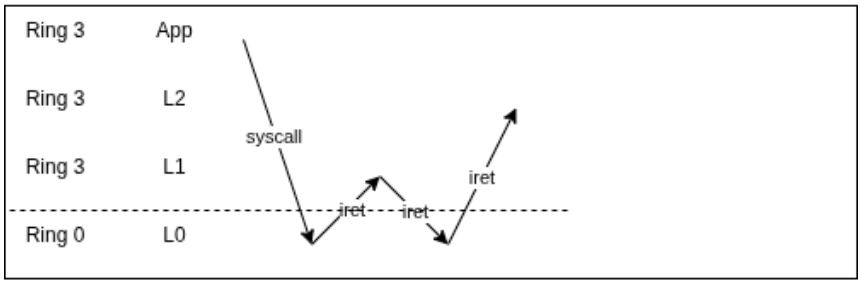
For a hardware VMM:

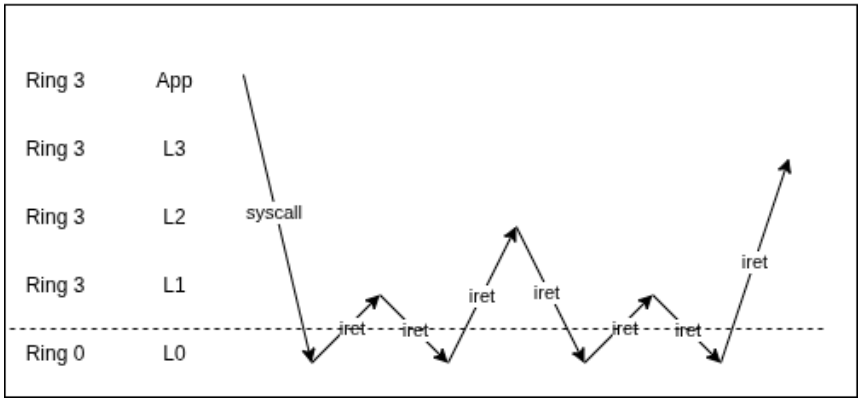
- The `syscall` instruction makes the process go from Ln's non-root ring 3 to Ln's non-root ring 0

For a software VMM:

- The `syscall` instruction has to be trapped by L_{n-1}, which has to then forward it to L_n
- So the number of privilege transitions involved is the number of privileged transitions that it takes to reflect a syscall from L_n to L_{n-1} plus the number of privilege transitions to reflect L_{n-1}
- So the number of privilege transitions is 2^n







Problem #9 (16 points)

File systems are commonly implemented as kernel drivers. This means that while they run in privileged mode, just like the kernel, they do not require any modification to the kernel itself. `xsyncfs` on the other hand relies on a modified Linux kernel called Speculator, which enables speculative execution ahead of file system operation completion. In the real world, neither the kernel modifications for Speculator nor the ones for Dune have made it into the mainline kernel, but for the sake of this question, let's assume that Dune, and only Dune, is now a feature of the mainline Linux kernel.

Could you implement external synchrony inside a Dune process without modifying the host Linux kernel?

ANSWER:
Speculator has to manipulate data structures that belong to the kernel in order to allow speculative execution ahead of syscall completion. The only way for a Dune process to provide control over these would be to emulate an entire "guest OS" and limiting the processes benefiting from speculation to communicating between each other rather than with the whole system. Dune would have to use a synchronous file system under the hood. So either Dune can be used to isolate processes which can use `xsyncfs`, but Dune cannot let processes use `xsyncfs` if they interact with anything outside of the Dune process.

Problem #10 (20 points)

The Aegis exokernel was implemented by the same authors for the MIPS architecture instead of the x86 architecture, prior to Xok. The authors compare Xok to Aegis in section 5.1. The MIPS architecture uses a software-managed TLB, meaning that TLB misses are handled by privileged software rather than hardware. The Aegis exokernel thus differed from the Xok exokernel in that it let unprivileged software write their own page tables, and it would only verify their validity upon TLB misses.

Say we want our exokernel to use superpages.

A. In the case of Xok and in the case of Aegis, should fragmentation be handled by the exokernel, by the libOS, or both?

ANSWER:
The answer has to be the same for both Xok and Aegis, as fragmentation is a matter of physical memory management, which is independent of TLB management/TLB misses/page tables. The philosophy of the exokernel is to handle protection and to let management and to let libOS's handle management. These responsibilities are incompatible for fragmentation: if the exokernel doesn't manage fragmentation, then libOS's might step on each other's feet and prevent each other from allocating contiguous memory regions.

B. In the case of Xok and in the case of Aegis, should promotion be handled by the exokernel, by the libOS, or both?

ANSWER:
The Xok exokernel has to provide an interface to promote pages into superpages because its libOS's cannot define their own page tables, but the libOS should be the one requesting promotions.
The Aegis exokernel doesn't have to do anything special to set up superpages, everything can (and should) be done in the libOS.

Problem #11 (36 points)

Consider a process that, just like in the reading question, and as described in the second paragraph of section 3.7, has some threads running in Dune mode and some regular Linux threads not running in Dune mode. The program running in this process is syscall intensive, therefore we decide to dedicate some threads whose only job is to request syscalls. When another thread needs to do a syscall, it will place somewhere in memory a syscall request which will be read by a syscall-executing thread.

A. Should the syscall intensive threads be running in Dune mode or outside of Dune mode, and why?

ANSWER:
The syscall intensive threads should be running outside of Dune me to avoid having to perform costly VMCALL/VMEXIT operations to get to the host Linux kernel.

B. Say a thread wants to call `read(int fd, char *buf, size_t count)`. This thread and the one that will request the syscall on its behalf may not execute in the same mode (cf previous question). Will a pointer to `buf` passed by the calling thread be valid for the syscall-executing thread as well?

ANSWER:
Dune's guest physical addresses match Linux's virtual addresses. If the Dune process uses these addresses, then they can be shared across the "Dune border". If the Dune process sets up EPTs that are not identity mapped, then pointers inside of Dune mode won't be valid outside of it (and vice versa).

… Problem #11 continued from previous page

C. Say that, for some reason, both the calling thread from the previous question and the syscall-executing thread try to access the array pointed to by `buf`. Would these memory accesses use the same TLB entry?

ANSWER:
They would be different entries because the Dune process's entry would be the Guest Virtual Address -> Host Physical Address translation whereas the non-Dune entry would be a Host Virtual Address -> Host Physical Address.

D. Say your application has 2 threads, and you decide to dedicate 1 of them to executing syscalls. The other thread of the application will push syscall requests to a shared queue. In parallel to the execution of a syscall by the syscall-executing thread, the "main" thread can keep running and push more syscall requests to that queue. When the syscall-executing thread completes the execution of a syscall, it pushes its return value on a queue that is consumed by the main thread. What structure/assumptions would lead to a livelock in this design?

ANSWER:
If

- there is a single core available,
- the syscall request queue is unbounded,
- and the syscall-requesting thread runs with a higher priority than the syscall-executing thread

Then a livelock will occur.
Other combinations of assumptions could lead to livelocks!