CS 212 Final Review

Post-midterm topics

Spring 2026

Admin

- Final is **cumulative** — covers the entire course
- Open note: bring printed lecture notes, no electronic devices
- Post questions on Ed under **Lectures+Exams**

Today: everything after the midterm (plus a refresher on linking).

Topics today

Covered in midterm review:

- Processes & threads
- Scheduling
- Concurrency / synchronization
- Virtual memory

Today (in this order):

- Virtual machines
- Protection
- Linking
- Networking
- File systems
- I/O & disks
- Memory allocation

Virtual Machines

One more level of indirection

OS vs. VMM

Both share hardware with protection. Different *abstractions*:

- OS exposes a **process** abstraction
  - Apps see syscalls, virtual memory, files
  - Makes finite resources look bigger
- **Virtual Machine Monitor (VMM)** exposes a **hardware** abstraction
  - Guest sees CPU, MMU, devices — as if running on bare metal
  - Lets you run arbitrary {OS + apps} on top
  - Isolation *between OSes*, not just apps

Benefits: software compatibility, consolidation, isolation, snapshotting.

Trap-and-emulate

Classical virtualization:

- Guest OS runs in *user* mode
- Privileged instructions trap into the VMM
- VMM emulates them, returns to guest

x86 problem: not classically virtualizable.

- Some sensitive instructions *don't* trap in user mode (e.g. `popf` silently ignores the IF flag)
- Guest can "see" it's not in ring 0

Two solutions:

- **Binary translation** (VMware, pre-VT-x)
- **Hardware-assisted virtualization** (Intel VT-x, AMD SVM)

Binary translation & hardware support

Binary translation:

- Dynamically rewrite guest kernel code
- Sensitive instructions replaced with VMM call-outs
- Most instructions copied verbatim (translation $\approx$ identity)
- Translated code can then safely run in kernel mode

Hardware-assisted (covered in lecture for AMD):

- New "guest mode"
- `vmrun` enters guest, loading state from VMCB
- VMCB control bits say which events cause a "VM exit"
- On exit, hardware saves state back to VMCB, VMM resumes

Memory virtualization

Now there are *two* levels of translation.

- Guest OS controls Guest VA $\rightarrow$ Guest PA (thinks it's physical)
- VMM controls Guest PA $\rightarrow$ Host PA

Hardware only does one translation. Two approaches:

- **Shadow page tables**: VMM constructs a page table mapping Guest VA $\rightarrow$ Host PA directly, kept in sync with the guest's
- **Nested page tables** (HW): hardware walks both, native two-level

Memory overcommit: ballooning

VMMs oversubscribe RAM: $\sum$(guest memory) $>$ host RAM. Works because most guests don't use all their RAM... until they do.

Problem: VMM wants to reclaim guest memory, but can't tell from outside which pages the guest considers "free."

Balloon driver (runs inside the guest):

- VMM asks the driver to "inflate" by N pages
- Driver `mallocs` N pages via the guest OS $\Rightarrow$ guest thinks they're used
- Driver reports those guest physical addresses to the VMM
- VMM unmaps them (Guest PA $\rightarrow$ Host PA) and gives the host RAM to someone else
- **Deflate**: VMM tells driver to free pages back; guest reuses them

ESX idle memory tax: bias reclamation toward pages the guest hasn't touched recently $\Rightarrow$ hot pages stay, cold pages get squeezed.

I/O virtualization

- Trap on guest's access to device registers
- Emulate device behavior in the VMM
- Forward to real device (or a virtual one, or a file)
- **Paravirtualization**: give the guest a special driver that talks to the VMM efficiently (virtio)
- **Pass-through / SR-IOV**: hardware exposes virtual devices directly to the guest

Example: shadow page tables

Q: Guest OS modifies its page table to map guest VA `0x1000` to guest PA `0x4000`. What does the VMM need to do?

Example: shadow page tables

Q: Guest OS modifies its page table to map guest VA `0x1000` to guest PA `0x4000`. What does the VMM need to do?

A: Detect the write (page table page is write-protected $\Rightarrow$ trap). Look up its own mapping: guest PA `0x4000` $\rightarrow$ host PA, say, `0x800000`. Install shadow entry: guest VA `0x1000` $\rightarrow$ host PA `0x800000`.

Example: shadow page tables

Q: Guest OS modifies its page table to map guest VA `0x1000` to guest PA `0x4000`. What does the VMM need to do?

A: Detect the write (page table page is write-protected $\Rightarrow$ trap). Look up its own mapping: guest PA `0x4000` $\rightarrow$ host PA, say, `0x800000`. Install shadow entry: guest VA `0x1000` $\rightarrow$ host PA `0x800000`.

Cost: every guest page-table edit is a trap. Hardware nested paging avoids this entirely.

Protection

Who can do what?

Access control models

Two ways to describe "who can access what":

- **Access Control List (ACL)**: per-*object* list of allowed subjects
  - Unix: each inode stores owner, group, perm bits for {user, group, other}
  - Easy to ask "who can access this file?"
- **Capabilities**: per-*subject* list of accessible objects
  - Process holds unforgeable tokens
  - Easy to ask "what can this process do?"
  - Process must *explicitly* invoke a capability → no "ambient authority"

setuid & its problems

- `setuid` bit: program runs with privileges of file's owner, not caller
- Necessary evil: e.g., `passwd` edits root-owned password file on your behalf
- Bug magnet:
  - Process inherits user's environment, fds, signals, cwd, ...
  - Any mistake $\Rightarrow$ privilege escalation
- **Confused deputy**: privileged program tricked into using its privileges on attacker's behalf
  - Capabilities help: process must name *which* authority to use

TOCTTOU

Time-**O**f-**C**heck **T**o **T**ime-**O**f-**U**se

```
if (access(path, R_OK) == 0) {      // check perms as real uid
    fd = open(path, O_RDONLY);      // open as effective uid
    ...
}
```

Q: What can the attacker do between the two lines?

TOCTTOU

Time-**O**f-**C**heck **T**o **T**ime-**O**f-**U**se

```
if (access(path, R_OK) == 0) {      // check perms as real uid
    fd = open(path, O_RDONLY);      // open as effective uid
    ...
}
```

Q: What can the attacker do between the two lines?

A: Replace `path` (e.g., a symlink they control) so the `access` check succeeds against one file but `open` opens *another*. Race window = privilege escalation.

TOCTTOU

Time-**O**f-**C**heck **T**o **T**ime-**O**f-**U**se

```
if (access(path, R_OK) == 0) {      // check perms as real uid
    fd = open(path, O_RDONLY);      // open as effective uid
    ...
}
```

Q: What can the attacker do between the two lines?

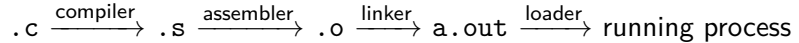
A: Replace `path` (e.g., a symlink they control) so the `access` check succeeds against one file but `open` opens *another*. Race window = privilege escalation.

Fix: drop privileges, or use `openat` + checks on the resulting fd, never re-resolve the name.

Linking

From source files to a running program

The pipeline



- **Compiler/assembler**: one `.o` per source file
  - Knows about its own file's symbols
  - Leaves external references unresolved (e.g., `printf`)
- **Linker**: combines `.o` files into one executable
  - Pass 1: coalesce segments, build global symbol table, assign virtual addresses
  - Pass 2: walk relocations, patch each reference with the right address
- **Loader**: reads executable, maps segments into the process's address space, jumps to entry point

Memory layout the linker has to build

- **.text**: code, read-only
- **.rodata**: constants, read-only
- **.data**: initialized globals, R/W
- **.bss**: zero-init globals, R/W (no file space!)
- Heap, stack: not laid out by linker

Who builds what?

- Heap: `malloc` at runtime
- Stack: compiler picks layout (relative to frame ptr)
- Globals: *compiler emits, linker lays out*
- Mmaped: programmer / dynamic linker

Binding time: *compile* $\rightarrow$ *link* $\rightarrow$ *load* $\rightarrow$ *runtime*. Linker resolves things known at link time.

Example: what does the linker do?

Two files:

- `main.c`: defines `main`, calls `foo()` and prints `x`
- `foo.c`: defines `foo` and `int x = 42;`

Q: In `main.o`, what does the call to `foo` look like? Where does `x`'s address come from?

Example: what does the linker do?

Two files:

- `main.c`: defines `main`, calls `foo()` and prints `x`
- `foo.c`: defines `foo` and `int x = 42;`

Q: In `main.o`, what does the call to `foo` look like? Where does `x`'s address come from?

A: `main.o` contains `call <foo>` with a placeholder offset, plus a *relocation entry* saying "patch this offset once you know where `foo` lives." Same for `x`. The linker assigns addresses to `foo` and `x` during pass 1, then patches the relocation in pass 2.

Static vs. dynamic linking

- **Static linking**: pull library code into the executable at link time
  - Self-contained, no runtime surprises
  - Every program ships its own copy of `libc`, etc. (wasteful)
  - Can't upgrade libraries without relinking
- **Dynamic linking**: resolve library references at *load* or *runtime*
  - One copy of `libc.so` on disk, shared in memory across processes
  - Library upgrades take effect immediately
  - Runtime failure modes: missing library, wrong symbol, ABI mismatch
  - No compile-time type checking of library calls

Dynamic shared libraries: PIC, PLT, GOT

Problem: shared library may be loaded at *any* virtual address, but we want to share one copy of its code between processes.

- **Position-independent code (PIC)**: code doesn't bake in absolute addresses
- **GOT** (Global Offset Table): per-process, R/W; holds addresses of external symbols
- **PLT** (Procedure Linkage Table): per-process, executable; stub trampolines that jump through GOT

Call to `printf` in PIC code:

1. `call printf@plt` (a stub in our PLT)
2. PLT entry: `jmp *GOT[printf]`
3. First call: GOT entry points back to PLT → runs `dlfixup` → resolves `printf`'s address → writes it into GOT
4. Future calls: GOT already filled → direct jump

This is lazy dynamic linking.

Linking & security

Loader/linker is the last chance to inject defenses:

- **W^X**: no page is both writable and executable
  - Stops classic "inject shellcode into buffer, return into it"
  - JITs must explicitly toggle perms
- **ASLR**: randomize segment addresses
  - Makes return-to-libc / ROP harder, not impossible
  - Needs PIC-style executables (`-fpie -pie`)
- Also (from the compiler side): stack canaries, CFI

Networking

Talking to other machines

Layering & encapsulation

- Application → Transport (TCP/UDP) → Network (IP) → Link (Ethernet)
- Each layer *wraps* the one above by prepending a header
- IP: unreliable, best-effort datagrams
- TCP, layered on IP, gives reliable byte streams: acks, retransmits, ordering, congestion control
- OS exposes a **socket**: bidirectional byte stream, like a pipe but across machines

Networked file systems (NFS)

Want: same FS interface, data could be on another machine.

- NFS: **vnode** interface — abstract VFS layer above individual FSes
- Each vnode op (`read`, `lookup`, ...) becomes an RPC
- Designed stateless: server doesn't remember client state across requests
  - Crash recovery: client just retries
- Operations are idempotent (mostly):
  - `read(offset, len)`: same answer twice
  - `remove`: second one fails ⇒ server may need to remember a little

Example: NFS retries

Q: Client sends `write(file, offset=100, data=...)`. Reply drops. Client resends. Why is this safe?

Example: NFS retries

Q: Client sends `write(file, offset=100, data=...)`. Reply drops. Client resends. Why is this safe?

A: The write is identified by `(file, offset, len)`, not "the next write". Re-applying the same write at the same offset is idempotent — second write overwrites with same data. If we instead used "append" semantics, the retry would double-write.

Example: NFS retries

Q: Client sends `write(file, offset=100, data=...)`. Reply drops. Client resends. Why is this safe?

A: The write is identified by `(file, offset, len)`, not "the next write". Re-applying the same write at the same offset is idempotent — second write overwrites with same data. If we instead used "append" semantics, the retry would double-write.

Design lesson: stateless + idempotent makes crash recovery almost trivial. State is what makes distributed systems hard.

File Systems

Names & bytes $\rightarrow$ disk blocks

Goals & workload observations

Filesystem must map `(name, offset)` $\rightarrow$ disk blocks, while:

- Optimizing for disk properties (locality, sequential)
- Surviving crashes without corruption

Empirical patterns it exploits:

- Blocks in a file are used together, sequentially
- Files in a directory are used together
- Most files are small...
- ... but most *disk space* is in large files
- Need good sequential *and* random access

Storage schemes

- **Contiguous / extent-based**
  - Fast, simple
  - External fragmentation; hard to grow
- **Linked** (block $\rightarrow$ next block pointer)
  - Easy to grow, good sequential access
  - Random access = pointer chasing on disk = disaster
- **FAT** — linked, but pointers in a separate table
  - Pointer chasing in memory; data blocks pure data
- **Indexed (inode)** — array of block pointers per file
  - Hierarchical: direct, indirect, double-indirect pointers
  - Asymmetric: small files cheap, large files still possible
  - This is Pintos Project 4!

Directories & links

- Directory = file with a special format: list of `(name, inode#)`
- Hierarchical name space; each process has a cwd
- **Hard link**: another name → same inode (refcount in inode)
  - Same FS only
  - Can't hard-link directories (would create cycles)
  - File "deleted" when last hard link removed *and* no open fds
- **Soft (symbolic) link**: a file whose contents are a path
  - Can cross file systems
  - Dangling links are fine (target may not exist)

FFS optimizations over the original Unix FS

Problems with original Unix FS:

- Tiny blocks (512B): too much indirection
- Poor clustering: blocks of a file scattered everywhere

FFS fixes:

- Larger blocks ($\geq$ 4 KB); split unused tail into *fragments* to avoid internal frag
- **Cylinder groups**: keep inode & its data in same cylinder group
- Sequential file blocks placed in sequential sectors
- Bitmap (not free list) of free blocks $\rightarrow$ better placement
- Async/journaled metadata; `/tmp` in memory

Crash consistency

Rule: losing recent data is OK; corruption is not.

- **fsck**: scan after crash, fix what's fixable — but you must guarantee the on-disk state is *always* fixable
- **Ordered updates**: write structure before pointer, null pointers before freeing target, don't null all pointers to a live object
- **Soft updates**: track dependencies, write in any order but *undo* uncommittable changes temporarily
- **Journaling**: log first, apply second
  - "Write-ahead log": append a record describing the operation, `fsync` the log, then apply
  - On crash: replay log from last checkpoint (idempotent ops)

Example: ordered updates

Creating a new file "foo" in directory `D`:

1. Allocate inode *I*
2. Initialize *I* on disk
3. Add entry `("foo", I)` to *D*

Q: Why must (2) happen before (3) on disk?

Example: ordered updates

Creating a new file "foo" in directory `D`:

1. Allocate inode *I*
2. Initialize *I* on disk
3. Add entry `("foo", I)` to *D*

Q: Why must (2) happen before (3) on disk?

A: If (3) is written but (2) isn't, after a crash the directory references garbage — corruption fsck can't repair. If only (2) is written, the inode is allocated but unreferenced — a leak fsck *can* fix.

Log-Structured FS (LFS)

Idea: turn *all* writes into one big sequential log.

- Disks fast at sequential writes, RAM cache absorbs reads
- Inodes themselves go in the log $\Rightarrow$ "inode map" to find them
- Old data accumulates as new versions get written
- Background **cleaner** compacts segments

Tradeoffs:

- Excellent write throughput, especially small random writes
- Cleaner overhead under high utilization
- Natural fit for SSDs (no in-place overwrite anyway)

I/O & Disks

Talking to slow hardware

Device communication

How does the CPU talk to a device?

- **Memory-mapped registers**: device registers appear at magic physical addresses; load/store works
- **Memory-mapped device memory**: the device exposes its own RAM (e.g., framebuffer)
- **Special instructions** (`inb`/`outb` on x86): historical, slow, clunky
- **DMA**: hand the device a buffer pointer; it reads/writes main memory directly while the CPU does other work

Polling vs. interrupts

- **Polling**: loop checking "are you done?"
  - Wastes CPU when device idle
  - Great when device is *constantly* ready (high arrival rate)
- **Interrupts**: device signals CPU when ready
  - CPU does other work in the meantime
  - Bad under high arrival rate $\Rightarrow$ livelock: drowning in interrupt handlers, no progress on real work
- **Best in practice**: *adaptive* — interrupt-driven when idle, switch to polling under load

Disk geometry & access cost

- **Sectors** are the unit of transfer (typically 512B or 4KB)
- Cost of an access $\approx$
  - *Seek* (arm to right track) — ms, $\sim$10ms full stroke
  - *Rotational latency* (wait for sector under head) — ms
  - *Transfer* — comparatively fast
- Implications:
  - Sequential I/O $\gg$ random I/O
  - Long seeks $\gg$ short seeks
  - File systems should lay out related data contiguously

Example: interrupt livelock

A NIC delivers 100,000 packets/sec. Each interrupt costs $\approx 5\,\mu$s of overhead (entry, dispatch, return) *before* the packet is actually processed.

Q: What fraction of CPU goes to interrupt overhead alone? What about at 1M pps?

Example: interrupt livelock

A NIC delivers 100,000 packets/sec. Each interrupt costs $\approx 5\,\mu$s of overhead (entry, dispatch, return) *before* the packet is actually processed.

Q: What fraction of CPU goes to interrupt overhead alone? What about at 1M pps?

A: $100{,}000 \times 5\,\mu\text{s} = 500$ ms per second $\Rightarrow$ 50% CPU burned on interrupt overhead, before doing any work. At 1M pps: 5 s of overhead per second $\Rightarrow$ impossible. System livelocks — every cycle handles interrupts, no progress on userspace.

Example: interrupt livelock

A NIC delivers 100,000 packets/sec. Each interrupt costs $\approx 5\,\mu$s of overhead (entry, dispatch, return) *before* the packet is actually processed.

Q: What fraction of CPU goes to interrupt overhead alone? What about at 1M pps?

A: $100{,}000 \times 5\,\mu\text{s} = 500$ ms per second $\Rightarrow$ 50% CPU burned on interrupt overhead, before doing any work. At 1M pps: 5 s of overhead per second $\Rightarrow$ impossible. System livelocks — every cycle handles interrupts, no progress on userspace.

Q: How does an adaptive scheme (e.g. Linux NAPI) avoid this?

Example: interrupt livelock

A NIC delivers 100,000 packets/sec. Each interrupt costs $\approx 5\,\mu$s of overhead (entry, dispatch, return) *before* the packet is actually processed.

Q: What fraction of CPU goes to interrupt overhead alone? What about at 1M pps?

A: $100{,}000 \times 5\,\mu\text{s} = 500$ ms per second $\Rightarrow$ 50% CPU burned on interrupt overhead, before doing any work. At 1M pps: 5 s of overhead per second $\Rightarrow$ impossible. System livelocks — every cycle handles interrupts, no progress on userspace.

Q: How does an adaptive scheme (e.g. Linux NAPI) avoid this?
A: First interrupt arrives $\Rightarrow$ *mask* further interrupts and switch to polling. Drain the queue. When empty, re-enable interrupts. Pays interrupt overhead once per *burst*, not per packet.

Memory Allocation

Managing the heap

Memory allocator: the problem

- Support arbitrary number, size, lifetime of allocations, can't move them
- Core fight: minimize fragmentation
  - Different sizes + lifetimes leave "holes"
  - No universal solution — workload matters
- Two flavors of fragmentation:
  - **External**: free space exists but not in usable chunks
  - **Internal**: rounded up to a size class, slack inside the block
- Program behavior: ramps, peaks, plateaus
  - Arena/region allocators exploit "peak" phases (free all at once)

Allocation strategies

- **Best fit** — smallest block that fits
  - Leaves tiny unusable slivers $\Rightarrow$ external fragmentation
- **Worst fit** — largest block
  - Quickly destroys all large blocks
- **First fit** — first block that fits
  - In practice, often as good or better than best fit; faster
- **Segregated free lists / slab** — one list per size class
  - Fast, low external frag, but internal frag from rounding
- **Buddy** — power-of-two split/merge
  - Fast coalescing, but internal frag (up to $2\times$)

Example: buddy allocator

64 KB heap, buddy allocator. Allocations round up to the next power of 2.

Q: You allocate 5 KB. What block size do you get? What's free afterward?

Example: buddy allocator

64 KB heap, buddy allocator. Allocations round up to the next power of 2.

Q: You allocate 5 KB. What block size do you get? What's free afterward?

A: 5 KB $\rightarrow$ 8 KB. Splits: 64 $\rightarrow$ 32+32 $\rightarrow$ 16+16+32 $\rightarrow$ 8+8+16+32. Take one 8. Free list: $\{8, 16, 32\}$. Internal frag: 3 KB.

Example: buddy allocator

64 KB heap, buddy allocator. Allocations round up to the next power of 2.

Q: You allocate 5 KB. What block size do you get? What's free afterward?

A: 5 KB $\rightarrow$ 8 KB. Splits: 64 $\rightarrow$ 32+32 $\rightarrow$ 16+16+32 $\rightarrow$ 8+8+16+32. Take one 8. Free list: $\{8, 16, 32\}$. Internal frag: 3 KB.

Q: You then free the 5 KB block. Does it coalesce?

Example: buddy allocator

64 KB heap, buddy allocator. Allocations round up to the next power of 2.

Q: You allocate 5 KB. What block size do you get? What's free afterward?

A: 5 KB $\rightarrow$ 8 KB. Splits: 64 $\rightarrow$ 32+32 $\rightarrow$ 16+16+32 $\rightarrow$ 8+8+16+32. Take one 8. Free list: $\{8, 16, 32\}$. Internal frag: 3 KB.

Q: You then free the 5 KB block. Does it coalesce?
A: Yes — its buddy (the other 8) is free, merge $\rightarrow$ 16. That 16's buddy is also free $\rightarrow$ 32. And again $\rightarrow$ 64. Heap fully free.

Example: buddy allocator

64 KB heap, buddy allocator. Allocations round up to the next power of 2.

Q: You allocate 5 KB. What block size do you get? What's free afterward?

A: 5 KB $\rightarrow$ 8 KB. Splits: 64 $\rightarrow$ 32+32 $\rightarrow$ 16+16+32 $\rightarrow$ 8+8+16+32. Take one 8. Free list: $\{8, 16, 32\}$. Internal frag: 3 KB.

Q: You then free the 5 KB block. Does it coalesce?
A: Yes — its buddy (the other 8) is free, merge $\rightarrow$ 16. That 16's buddy is also free $\rightarrow$ 32. And again $\rightarrow$ 64. Heap fully free.

Pattern: coalescing is $O(1)$ per level, but only fires when *both* buddies are free simultaneously. Alternating allocation patterns can defeat it.

Garbage collection

- **Reference counting**
  - Each object tracks its refcount; free at 0
  - Simple, incremental
  - Cycles leak — two objects pointing at each other never reach 0
  - Refcount updates are expensive (atomic on shared objects)
- **Mark & sweep**
  - Walk from roots, mark live, sweep dead
  - Handles cycles
- **Stop & copy** (semi-space)
  - Copy live objects to new space; compacts memory
  - Half of heap unused at any time
  - Optimization: *concurrent* collection using page-faults — mutator runs, mark pages read-only, fault when mutator touches an unscanned page

Example: refcount vs. mark-and-sweep

Q: You have a doubly-linked list of 1000 nodes. The head pointer goes out of scope. What happens under refcount? Under mark/sweep?

Example: refcount vs. mark-and-sweep

Q: You have a doubly-linked list of 1000 nodes. The head pointer goes out of scope. What happens under refcount? Under mark/sweep?

A: Refcount: each node still has its prev/next neighbors pointing at it, so refcount ≥ 1 everywhere. Nothing is freed — leak. Mark/sweep: roots no longer reach the list, so the whole list is unmarked and reclaimed.

Good luck!

Questions?