EECS-343 Operating Systems
Lecture 19:
Final Review

Steve Tarzia

Spring 2019

Northwestern

Announcements

- HW4 is due on Friday and **cannot** be turned in late.
  - Answers will be posted on Saturday.

- Final exam is on Monday, 3-5pm.
  - Bring a calculator.
  - Norris bookstore sells calculators for $4.

Roles of an OS

- A ***user interface*** for humans to run programs
- A ***resource manager*** allowing multiple programs to share one set of hardware.
- A ***programming interface*** (API) for programs to access the hardware and other services.

Processes & System Calls

- *Process* is a program in execution
- *Limited direct execution* is a strategy whereby a process usually operates as if it has full use of the CPU & memory.
- CPUs have user and kernel *modes* to prevent user processes from running privileged instructions, thus *limiting* execution.
- *Interrupts* are events that cause the kernel to run
- *System Calls* (or traps) are software interrupts called by a user program to ask the OS to do something on its behalf.
- *Timer Interrupt* ensures that the kernel eventually gets a chance to run

Processes & System Calls

- **Process** is a program in execution
- **Limited direct execution** is a strategy whereby a process usually operates as if it has full use of the CPU & memory.
- CPUs have user and kernel **modes** to prevent user processes from running privileged instructions, thus *limiting* execution.
- **Interrupts** are events that cause the kernel to run
- **System Calls** (or traps) are software interrupts called by a user program to ask the OS to do something on its behalf.
- **Timer Interrupt** ensures that the kernel eventually runs.

Process Creation & Memory Layout

- Variations in CPU architecture influence OS design
- Linux supports 31 different CPU architectures
  - Low-level *mechanisms* are different on each arch.
  - High-level *policies* are the same for all.
- *Fork* syscall: run one, exits twice!
- *Nondeterminism* is when a program's output is unpredictable
- OS process scheduler can create *race conditions* in programs that rely on an interaction of multiple processes.
  - These are tricky to debug, because they are sensitive to timing (*Heisenbugs*).
- *Kernel panic* occurs when OS causes an exception and can't recover

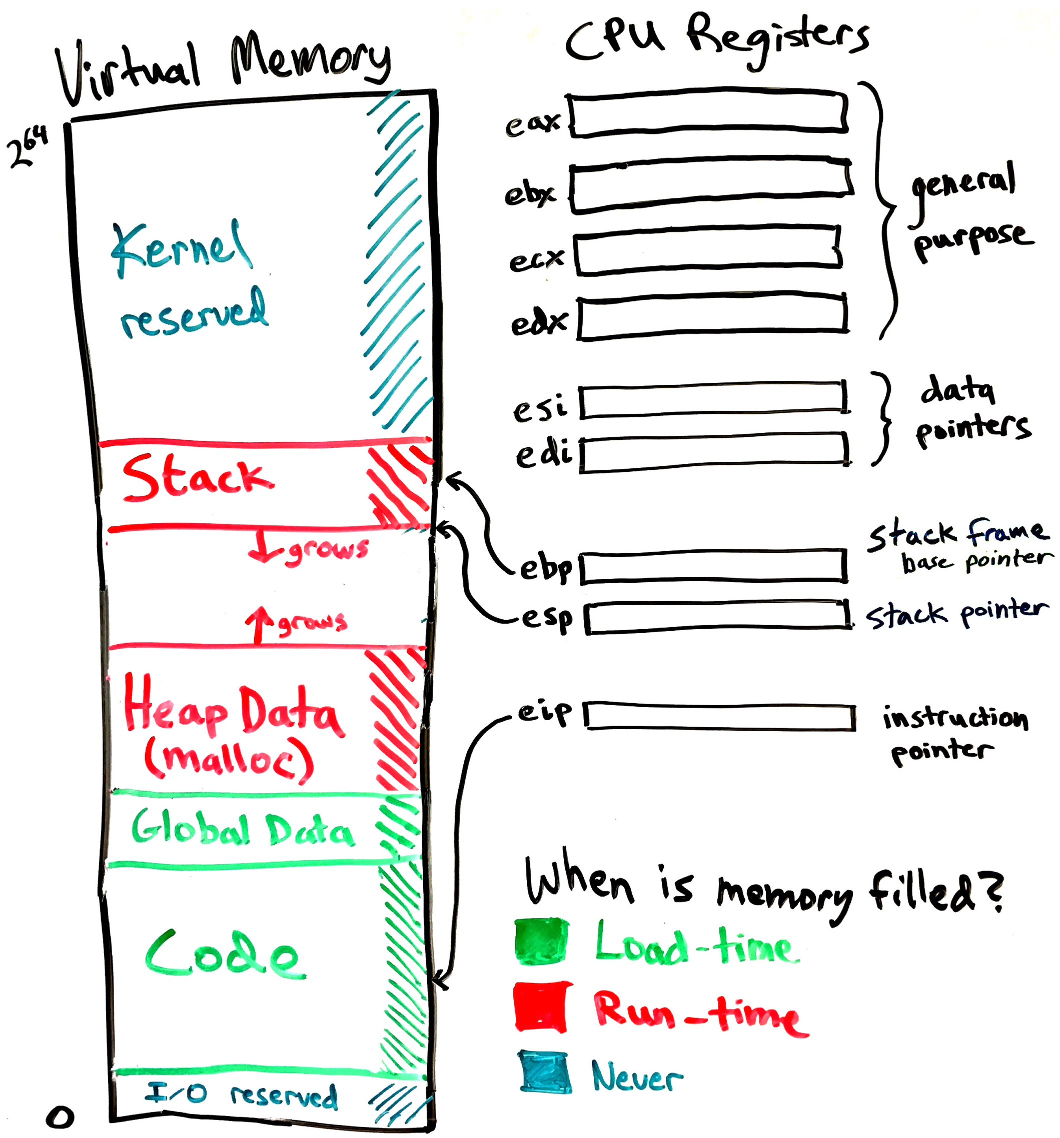
Process Memory & Virtual Memory

- Showed program's view of computer in more detail
- Explained how execution *stack* makes subroutines easy
- *Heap* is used by malloc to dynamically allocate memory
- *Virtual Memory* allows each process to have its own view of memory
    - Memory is divided into *pages*
- OS creates a *Page table* for each process
    - Tells CPU how to translate virtual to physical addresses
    - Page tables are the *mechanism* controlled by the OS to distribute physical memory among competing processes
    - OS can just change the CPU's `%CR3` register to change page tables.

Scheduling

- Defined two conflicting metrics: ***turnaround time*** and ***response time***
  - Cannot optimize both – must tradeoff, or balance, the two
- Optimized by ***shortest job first*** and ***round robin***, respectively
- Context switching overhead is due to the CPU caches
  - CPU keeps most recently used data in nearby caches, so it's more efficient to let an ongoing process continue.
- ***I/O-blocked*** processes make progress without using the CPU
  - We should prioritize I/O-bound processes
- ***Multi-Level Feedback Queues*** are often used in real OS schedulers
  - Prioritizes "polite" processes that use little CPU time when scheduled
  - CPU-bound processes squander their time quotas and lose priority

Process' view of memory



- Code and global data are filled by *exec* syscall to load a program.
- A new *frame* is pushed on the stack whenever a function is called. (And popped on return.)
- Heap data is managed by malloc

Virtual Memory

- Memory is divided into equal-sized ***pages***.
- ***Page tables*** translate virtual page numbers to physical page numbers.
- Showed the details of page table entries (PTEs):
  - High bits translate from virtual page number to physical page number.
  - Low bits in the PTE are used to indicate present/rw/kernel page.
- During a context switch, kernel changes the **%CR3** register to switch from the page table (VM mapping) of one process to another.
- VM is handled by both the OS and CPU:
  - **OS** sets up the page tables and handles exceptions (page faults).
  - **CPU** automatically translates every memory access in the program from virtual addresses to physical addresses by checking (*walking*) the page table.

VM & Paging optimizations

- **Latency cost**, because each memory access must be translated.
  - **Translation lookaside buffer (TLB)** caches recent virtual to physical page number translations.
  - Software-controlled paging removes page tables from the CPU spec and lets OS handle translations in software, in response to TLB miss exceptions.
- **Space cost**, due to storing a page table for each process.
  - Linear (one-level) page tables are large.
  - Smaller pages lead to less wasted space during allocation, but more space is consumed by page tables.
  - **Multi-level page tables** are the only way to truly conserve space.
  - Mixed-size pages reduce TLB misses.
- Copy-on-write fork, demand zeroing, lazy loading, and library sharing all reduce physical memory demands.

Swapping

- Disk is slow, but large, and can be used to store RAM's overflow
  - Disks have high ***throughput*** (transfer bitrate) but high ***latency*** (delay)
  - Magnetic disks have even higher latency than SSDs, due to moving parts.
- Paging and swapping work together, using the same CPU mechanisms
  - If a page is marked "not present" it may be either invalid or swapped to disk.
    - Or it might indicate *lazy allocation*, *lazy loading*, or *copy-on-write*, as we saw last time.
  - High bits of page table entry can store disk location of swapped page.
- ***Page replacement policy*** decides which page(s) to ***evict*** to free memory
  - Swapping can be done ***on demand*** or in the ***background***
  - Having some free physical frames will prevent delays for allocations.
  - ***Accessed bit*** and ***Dirty bit*** in PTEs inform the page replacement policy
- ***Thrashing*** is when swapping prevents the system from doing any work.
- ***Unified page cache*** handles both traditional paging and ***file caching***.
  - Makes filesystem access seem just as fast as memory access.

Types of page faults

- **Minor/soft**: Page is loaded in memory, but PTE is not configured:
  - OS just wants to be informed when the page is accessed, so it *pretends* to evict the page (just mark it *not present*). Useful if CPU has no accessed/dirty bit.
  - Memory can be shared from another process (eg., copy on write)

  Response: update the PTE.
- **Major/hard**: A disk access will be needed:
  - Anonymous page (process data) may have been swapped out.
  - Lazy-loading program executable.

  Response: load the page from disk
- **Invalid**: User program misbehaved:
  - Dereference null or invalid pointer.
  - Write to page that is read-only.
  - Execute code on a page that is not executable (for security).

  Response: terminate the process.

Free Lists

- Handled by libc's ***malloc*** and ***free***
  - Malloc uses ***sbrk*** or ***mmap*** syscalls
- Freed memory is put on a ***free list*** to be reused for later allocations.
- A single header can be cleverly used and re-used for two purposes:
  - As a linked list node when the block is free/available
  - To store the size of the allocated block to help service *free* calls.
- Free space management ***policy*** determines:
  - which free blocks to choose for an allocation, and
  - When to ***coalesce*** (join) adjacent free blocks
- Free block choice policies include:
  - **First**, **next**, **best**, and **worst** fit.

Threads

- Processes can have multiple ***threads*** sharing the virtual address space
- ***Critical sections*** are block of code that must be run ***atomically***
- If unprotected, critical sections lead to ***race conditions*** that make code ***indeterminant*** – we get different results depending on timing.
- ***Locks*** are the simplest ***mutual exclusion primitive***, with two main functions:
  - **Acquire/lock** – get exclusive access to a shared resource.
  - **Release/unlock** – release the shared resource.
- Concurrency occurs naturally in multi-CPU systems
- Concurrency is created by the process scheduler in single-CPU systems

Implementing Locks

- Hardware support for atomicity:
  - Disable interrupts
  - ***Test and set***
  - ***Compare and swap***
  - ***Fetch and add***
  - ***Load-linked*** & ***Store-conditional***
- Various lock implementations
  - Spinlock
  - Ticket lock
  - Yielding lock
  - Queuing locks
    - ***Park/unpark*** on Solaris
    - ***Futex*** on Linux

- Sophisticated locks can be more ***fair*** and avoid starvation, but they can add unnecessary context-switch overhead on multiprocessors.
- ***Two-phase locks*** try to combine the best of both approaches.
- OS scheduler and concurrent user code must coordinate for best performance.

Concurrent Data Structures

- Simplest strategy is to use ***one big lock***, but this limits concurrency
  - It's ***thread-safe***, but not really concurrent
- Concurrent queue used two locks (head & tail)
- Concurrent hash table used one lock per bucket
- ***Condition Variables*** are used to order threads, using *signal()* & *wait()*.
  - ***Wait*** puts a thread to sleep, ***signal*** wakes a waiting thread.
  - Pthreads allows ***spurious wakeups***, so we still need to check a status variable.
  - ***broadcast()*** wakes all waiting threads
- ***Producer/consumer queue*** was implemented using two condition variables.

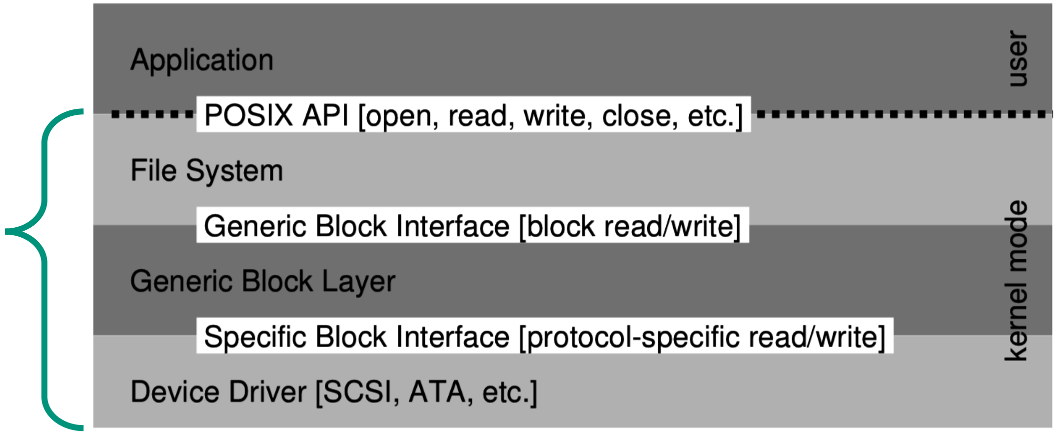
Synchronization Bugs

- ***Semaphore*** (up/down) is an all-purpose synchronization primitive
- ***Reader-writer*** lock allows multiple readers, but one writer.
- Adding too many locks can lead to ***deadlock***, which requires:
  - Mutual exclusion (avoid locks to avoid deadlock)
  - Hold and wait (use ***trylock*** to release first lock to before deadlocking)
  - No preemption
  - Circular wait (always acquire locks in the same order to avoid deadlock)
- Dining philosophers was an example of deadlock
  - Circular wait can be avoided by making one philosopher grab right-hand side instead of left first.

I/O and Disks

- OS interacts with devices by reading/writing ***device registers***
  - Each register has an ***I/O port*** address for in/out instructions, or
  - ***memory-mapped I/O*** uses special physical memory addresses (with `mov`)

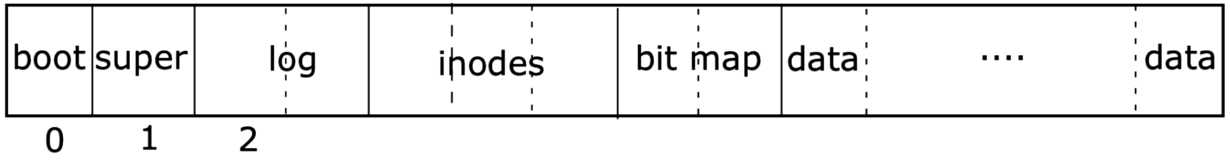
- Storage is complex, so kernel functionality is divided into at least three layers:



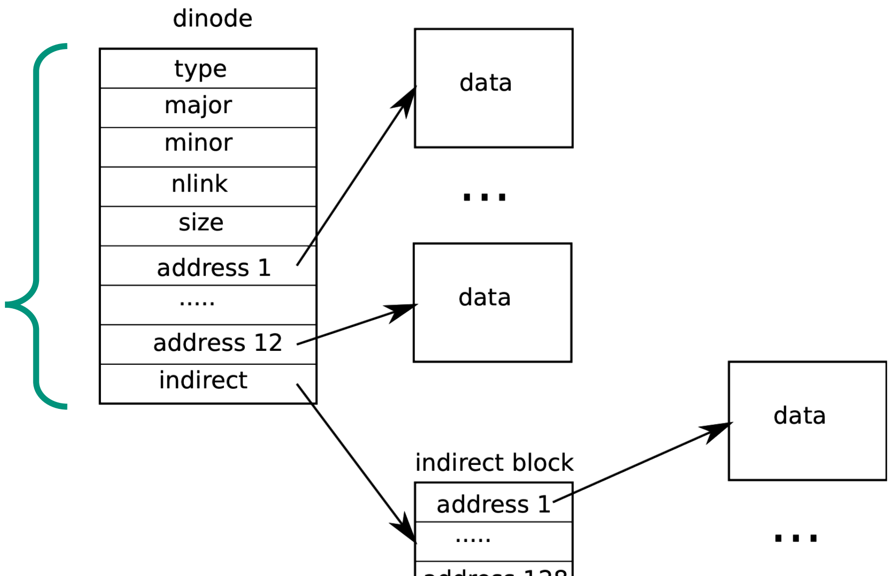
- Random access to a magnetic disk is 1000x slower than sequential
  - Read head must ***seek*** and disk must ***rotate*** to reach a new sector

RAID & File Systems

- RAID allows multiple disks to act together for better throughput, capacity, and/or fault tolerance.
  - ***Parity*** is used in ***RAID5*** to achieve all of the above.
- OSes have a application-level API (syscalls) for file I/O:
  - open, read, write, seek, stat, fsync, rename, unlink, mkdir
- ***Filesystem*** is a data structure the OS uses to organize disk space.



- Each file/directory has an ***inode*** storing metadata & pointers to data blocks.

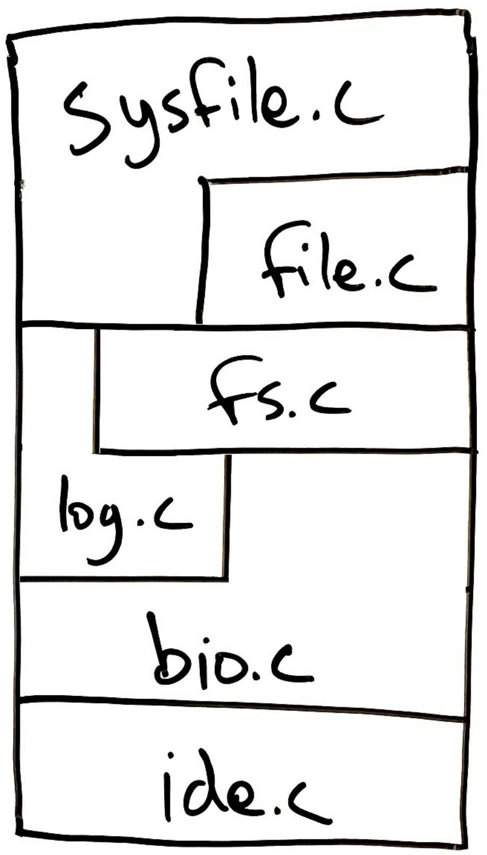


Buffer Caching & Logging

- Trace of file operations shows that many accesses to disk are needed for even a single open/read/write.
- To improve performance, ***cache*** a small number of active disk blocks
  - Allows later reads to happen in memory
  - Multiple writes can be absorbed and all are immediately visible in memory
- Each buffer is locked by a thread before use
- ***Write-ahead logging*** makes multiple disk writes appear atomic, even if the machine is powered-down in the middle of the transaction.
  - Very important for related changes to inodes & bitmap (metadata in general)
  - Data is written twice: to log first, then to main disk.
  - On reboot, interrupted transaction is either ***rolled back*** or ***replayed***.

Storage Layer Interactions

- Showed layered design of xv6 storage system
- Implementation of each layer uses only the layer(s) directly below
  - Must provide an API suitable for implementing the layer(s) directly above
  - Deeper layer are hidden.
- **`defs.h`** makes a subset of kernel functions in each file "public."
- Linux has a virtual file system (VFS) layer that allows multiple filesystems to coexist in one machine.



Log-structured File System

- Tries to make all writes ***sequential***, at the end of the disk (at first).
- ***Never edit*** data blocks or inodes, just write new copies and stop referring to the old versions. Inodes are scattered throughout the disk.
- ***Checkpoint region*** points to distributed inode map, to find inodes.
    - CR is the only thing that is always written in a well-known location.
    - Using an old version of the checkpoint region lets us see the filesystem as it looked in the past. LFS can be extended easily to become a *versioned file system*.
- ***Garbage collector*** occasionally scans FS to ***compact*** segments with old, unused versions of blocks.
- Restart from start of disk after reaching the end, filling in holes.


Memory Lane

Themes:

- *Hardware* provides features beyond basic C functionality to support OS
- *Virtualization*:
  - Providing processes with a simplified view of the underlying system
- *Caching & buffering* improves performance, assuming:
  - Temporal and/or spatial locality
- *Safely sharing resources*:
  - Data structures to quickly find unused resources and prevent accidental reuse
  - Policies to allocate resources among competing clients.
- *Concurrency is hard*: race conditions lead to bugs that are difficult to test.
- *Laziness*: when possible, delay handling requests if it may be possible to take a shortcut later or to amortize the cost of multiple requests.

Hardware features for OS use

- Privileged/kernel and user mode.
  - Privileged instructions.
- Interrupts:
  - Kernel specifies that CPU should jump to certain code to handle interrupts
  - Software interrupts (traps) can be initiated by user code for syscalls
  - Programmable timer and timer interrupts.
- Page table can be configured by OS for CPU to use virtual memory
  - PTEs can be marked "not present" or "read only" to implement:
    - Swapping, lazy loading, lazy allocation, copy on write
  - TLB makes VM efficient, and is sometimes managed directly by the OS.
- Atomic primitives to implement locks and lock-free synchronization
- In/out instructions and memory mapping to perform I/O

Virtualization/abstraction

- Limited direct execution temporarily gives processes full use of CPU (limited to non-privileged instructions)
  - This makes is very easy to write programs and compilers.
- Syscall interface hides hardware details and variations from processes
  - Eg., open("/home/steve/file.txt", READ|WRITE)
- Same program binary can be run on machines with different hardware, as long as the OS interface is the same.
  - *Application Binary Interface* (A**B**I) defines low-level OS-process-lib interactions.
    - ABI defines the Syscall numbers and the parameters for each syscall.
  - If OS provides same A**P**I (syscall/library function prototypes in header files), program source code need not change, but may be require recompilation.

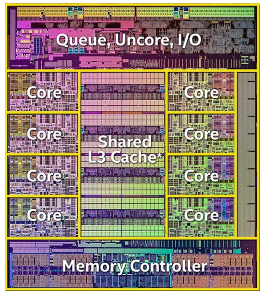
Caching makes data access faster

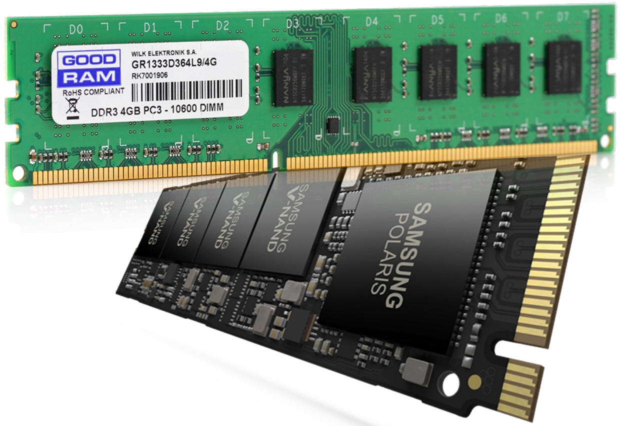
- Computers storage hierarchy has vastly different capacities ($10^{10}\times$) and latencies ($10^{7}\times$). Must choose small-and-fast or big-and-slow.
- Software and hardware should both be designed to take advantage of:
  - *Temporal locality*: access the same data frequently
    - Hardware caching (invisible to the OS) makes repeated access fast.
    - Software should be written and compiled to reuse memory locations.
  - *Spatial locality*: access *nearby* data frequently
    - Hardware caching pulls chunks of data into caches, so nearby values are on hand
    - Software should be written and compiled to move sequentially through data.
    - Rotating disks are especially sensitive to random vs sequential access.

Computers have a hierarchy of storage

Larger, but slower

delay		*capacity*
0.3ns	CPU Registers	1 kB (kilobyte)
5ns	CPU Caches (L2)	16 MB
50ns	Random Access Memory (RAM)	16 GB
100µs	Flash Storage (SSD)	1 TB
5ms	Magnetic Disk	8 TB





- Disk is about *ten billion* times larger than registers, but has about *ten million* times larger delay (latency).
- Goal is to work as much as possible in the top levels.
- Large, rarely-needed data is stored at the bottom level
- "Memory" is not just RAM, but everything below the registers

Cache and buffer examples

- Virtual memory → physical memory
- Physical memory → L1/L2/L3 cache
- Physical memory → CPU registers (managed by compiler)
- Magnetic disk sectors → "disk buffer" (on disk, hidden from OS)
- Magnetic disk sectors → solid state memory (in a *hybrid* or *fusion* disk)
- Disk sectors → buffer cache in RAM
- On-disk inodes → inode cache in RAM

Resource sharing

- CPU
  - Various scheduling policies (MLFQ, etc.)
  - *Mechanism* is the interrupt.
- Physical memory
  - Eviction policy for swapping (LRU, etc.)
  - *Mechanism* is paging.
- Persistent storage
  - Generally don't place quotas, but limit access to files according to owner.

Laziness is a virtue

Main idea is that programs often ask the OS to do unnecessary work.

- Fork: copy page table to child and mark everything read-only.
  - Make true copies of memory pages only in response to page faults.
- Mmap & sbrk:
  - don't have to write zeros to new memory or reserve space in physical memory until process actually uses it.
- Buffer cache: on `bwrite` we have two options:
  - *Write through*: write to disk immediately (in xv6)
  - *Write back*: wait to write until buffer is evicted from cache (the lazy approach).
- LFS: write a full segment of updates at ones (buffering helps)

Performance lessons

- My program is slow… why?
- Maybe I need a theoretically faster algorithm or data structure, … or maybe there is a system-level issue:
  - Programs may be using more memory than is physically available and thus are doing a lot of swapping (*thrashing*).
  - Another process may be using lots of memory or CPU time (these are shared resources). Important program may have insufficient *priority* in scheduler.
  - Process may be reading or writing a lot from *disk*.
  - There may be lots of CPU activity on one thread. Somehow dividing the work among *many threads* would speed things up (but then we need to worry about concurrency bugs).
- Much can be learned from simple tools like "top" and Task Manager.

Debugging lessons

- Often, SW engineers spend more time *reading* than writing code.
  - So, made your code readable for the next developer!
- If your code doesn't work, it's often helpful to:
  - Know exactly what changes you have made (git diff)
  - Maybe throw out your changes and start again (*roll back*)
- Unpredictable bugs are often due to *race conditions*.
  - Must protect critical sections and enforce ordering where necessary.

Reliability and Security lessons

- Disks are very prone to failure, but RAID significantly reduces the likelihood of losing data (or experiencing *downtime*).
- Users are prone to do dumb things like "`sudo rm -Rf /`"
  A *versioned*, log-structured filesystem lets you travel back in time to see what the filesystem used to look like.
- Untrustworthy apps may try to violate your privacy and sabotage your system. Proper isolation of processes and filesystem permissions can reduce the possible damage.

Meltdown and Spectre hacks

- OS relies of cooperation of software and hardware engineers.
- Bad things can happen when SW and HW engineers don't work together and understand each others' work.

- Huge vulnerabilities were discovered in 2017:
  - **Meltdown** – leaks protected memory through *out-of-order execution*.
  - **Spectre** – leaks protected memory through *speculative execution*.
- https://youtu.be/RbHbFkh6eeE

Meltdown

Problem: Attacker can influence speculative control flow

Bug: Speculative execution not subject to page permission checks.

Permissions are checked before exposing results, but cache can be affected and thus memory access timing leaks information.

Result: User code can read kernel data (secret), or another process' data.

Three steps:

1. *Setup*: flush the cache
2. *Transmit*: force speculation that depends on secret
3. *Receive*: measure cache timings

Meltdown overview

Initial setup:

```
char* kernel_addr = 0xFFFF0000; // The target is a high VM address in kernel space.
char probe[256 * 4096]; // The speed of access to this userspace array will leak data!
```

Guess the value of that kernel byte:

```
char guess = 0x00; // maybe 0x0 is stored in location 0xFFFF0000?
clflush(probe[guess * 4096]); // flush cache for the page corresponding to our guess.
```

Use kernel data cleverly in an instruction that will be rolled-back:

```
*0; // Generate an exception, for example dereference a null pointer.
probe[*kernel_addr * 4096]; // secret kernel data will control which of our array pages is accessed.
            // This should not even be executed, and if executed it should generate a page fault,
            // but it will be speculatively executed and discarded (but cache warming is leaked!).
```

Measure the side effects on the cache:

```
s = rdtsc(); // start timing
probe[guess * 4096];
e = rdtscp(); // end timing
if (e - s < CACHE_MISS_THRESHOLD)
  printf("guess was right!\n");
```

…**else**: Repeat with another guess!

From https://meltdownattack.com/meltdown.pdf

```
raise_exception();
// the line below is never reached
access(probe_array[data * 4096]);
```

Listing 1: A toy example to illustrate side-effects of out-of-order execution.

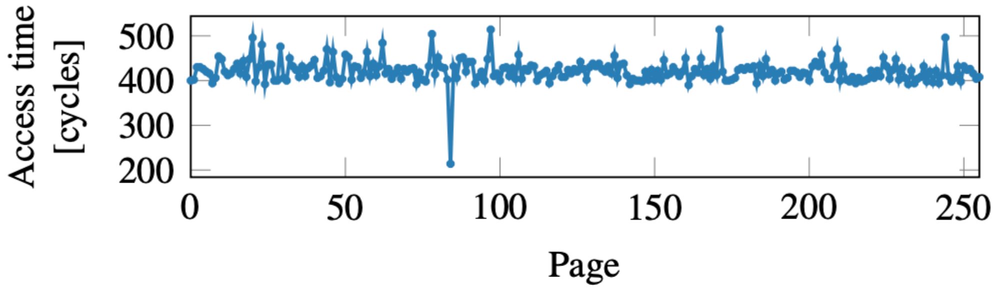


Figure 4: Even if a memory location is only accessed during out-of-order execution, it remains cached. Iterating over the 256 pages of `probe_array` shows one cache hit, exactly on the page that was accessed during the out-of-order execution.

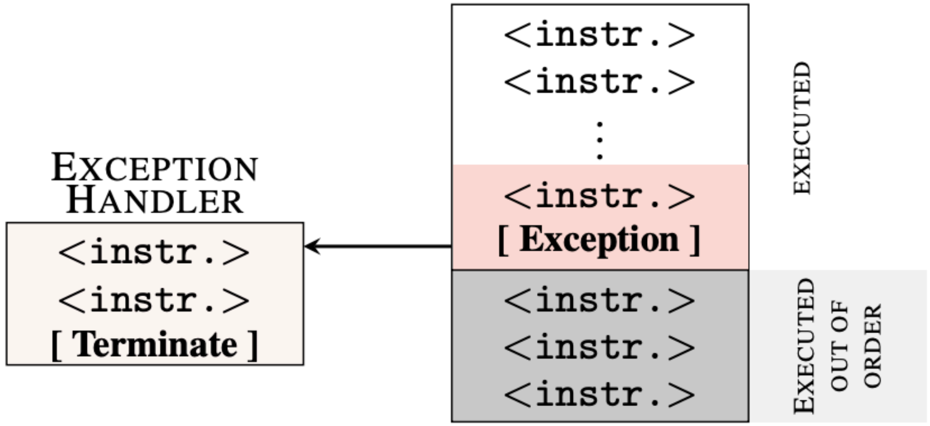


Figure 3: If an executed instruction causes an exception, diverting the control flow to an exception handler, the subsequent instruction must not be executed. Due to out-of-order execution, the subsequent instructions may already have been partially executed, but not retired. However, architectural effects of the execution are discarded.

Spectre

- **Problem**: Attacker can influence speculative control flow (same as Meltdown)
- **Attack**: Extract secrets within a process address space (e.g. a web browser). Can also be used to attack the kernel.
- Could use attacker provided code (JIT) or could co-opt existing program code
- Same basic three steps! Different setup and tester.

- Uses **branch prediction** cache instead of memory cache to leak effects of the rolled-back instructions.

Followup classes

If you enjoyed this class, then you should consider:

- 446 Kernel and other low-level software development
- 354 Network Penetration and Security
- 397 Digital Forensics
- 340 Intro to Computer Networking
- 339 Databases
- 203 Intro to Computer Engineering (fulfills basic engineering req.)
- 396 Scalable Software Architectures
- COMP_ENG 361 Computer Architecture I
- COMP_ENG 358 Intro to Parallel Computing

Questions?

Career advice!

- Software systems are affecting society drastically. **Take responsibility**!
- You'll be well paid, but don't spend it all. **Avoid golden handcuffs**!
  - This will give you the freedom to quit if you need to, and maybe start your own company.
- Software engineering is difficult and a lifelong learning experience.
  - Your degree is just the start of a very long path.
  - It doesn't matter too much where you start, as long as you're learning.