CS 350 - W24
Operating Systems
Full Course Notes

With Prof Bernard Wong

These are my in-class lecture notes. They cover absolutely everything besides example problems.

[Josiah Plett]

CS 350 Notes [1]

1) Intro Lecture

~~Preemption: take resources back~~

- The class just switched from the Harvard OS161 to ours.

Purpose of OS ★

- Make development hardware-agnostic.
- Provide abstraction + protection.
- Resource management.

"Library"

- Primitive OS: 1 program at a time, assumes no bad programs.
  - → bad utilization of hardware and of users.
- Multitasking OS: >1 process at a time, whenever one blocks.
  - → infinite loops, overwrite memory
- Multi-user OS: Use protection, management (won't be N times larger / slower!)
  - → gluttonous users, not enough memory, privacy, speed.

2) Protection, Structure, Processes

Ways to Protect ★

- Pre-emption: take resources away (save values)
- Interposition/Mediation: OS checks the legality of every access
- Privilege modes in CPUs: protected operations need privilege

Apps are underprivileged (user), OS is privileged (kernel)

OS Structure (typical)

USER
P1 P2 P3

KERNEL
VM file system
Drivers
network console disk
sockets
scheduler

OS Kernel → creates/deletes processes
→ provide hardware access

To allow apps to have more privilege:

System Call Handler

Which is a list of commands made by the OS to interact with sys & hardware.

TRAP

Processes ★

Process: An instance of a program running.

Executable file: An image of a process (needs to be loaded).

Why Processes? → simplicity of programming; abstraction!
→ higher throughput, lower latency.

3) Views of Processes

A Process's View ★

Each process has:
- an address space →
- its own open files
- its own virtual CPU

stack
↓
↑
heap
data
text (code)

A User's View ★

Creating Processes ⇨
- `int fork(void)` → creates exact copy process
  → returns ID of new process in parent (zero/0 otherwise)
- `int waitpid(int pid, int *stat, int opt)` → wait for child process (by pid) to terminate. Returns pid, or -1 on error.
  → stat contains exit value or signal → opt is usually 0 or WNOHANG

Deleting Processes ⇨
- `void exit(int status)` → cease current process.
  → status shows up in waitpid (shifted); 0-success, !0-error
- `int kill(int pid, int sig)` → sends sig to process pid
  → SIGTERM: kills but can be caught. (for cleanup like saving) SIGKILL: always kills

Running Programs ⇨
- `int execve(char *prog, char **argv, char **envp)` → execute a new program
  → usually called through wrappers
  → replaces current process!
  - prog – full pathname
  - argv – arg vector for main
  - envp – environment vars, eg. PATH, HOME

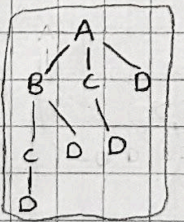
File decriptor: (aka file handle)

An index in our process' "Open File Table."

4) Other functions + Kernel's View of Processes

```
int main() { //A
  fork();  // B
  fork();  // C
  fork();  // D
  return 0;
}
```

⇨



Manipulating File Descriptors ⇨ `int dup2(int oldfd, int newfd)` → Closes newfd
→ copys oldfd into newfd (deep)

Pipes ⇨ `int pipe(int fds[2])` → two file descriptors, fds[1] fds[0]
(0 on success, 1 on error) → writes to fds[1] can be read on fds[0]

Operations on Pipes ⇨ read, write, close → when fds[1] closed, read(fds[0]) returns 0

eg: $ command1 | command 2
fds[1]   fds[0]

⑤ Kernel view of Processes

Why Fork?

Simplicity of interface: no args at all!

Quick to use: eg. pre-forking webservers

Lanched easily: execve or spawn

Scheduling

How to pick which process to run.

First runnable? FIFO/FILO? Priority?

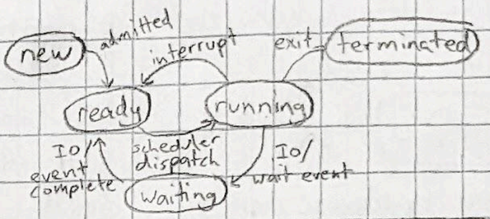
Preemption: give control back to kernel.

Kernel Implementing Processes

Process state
Process ID
User id, etc.
Program counter
Registers
Address space (VM)
open files

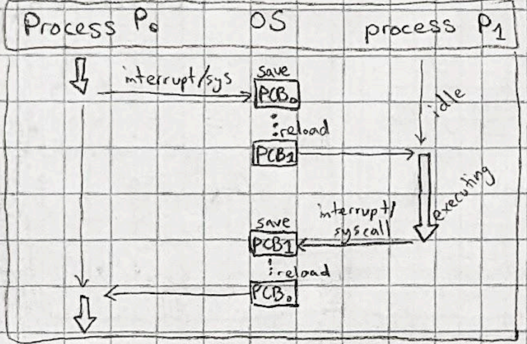
PCB

- OS keeps data structure for each proc, "PCB" (Process Control Block)
- Tracks state of process (running, ready, blocked)
- Includes info necessary to run (registers, etc.)



Context Switching

Context Switch: running process changes.



DETAILS
- Save program counter/registers
- Save condition codes
- Change virtual address locations
- May require flushing TLB
- May have non-negligable implications on cost due to more cache misses.

⑥ Context Switching + Threads

Threads

A thread is a schedulable execution context.

Multithreaded programs share the same address space.

BENEFITS
- Great abstraction for concurrency (lighter weight than processes)
- Allows 1 process to use many CPUs.
- Allows program to overlap I/O and computation.
- Kernels can have their own threads, too.

POSIX Thread API

int pthread_create(...) → creates new thread w/attributes, run fn w/args

void pthread_exit(...) → destroy current thread & return pointer

int pthread_join(...) → wait for thread to exit and recieve return v.

void pthread_yield() → tell the OS scheduler to run a diff thread.

Kernel thread: not necessarily in the kernel; just scheduled by the kernel.

User thread: created/scheduled by a user's thread library.

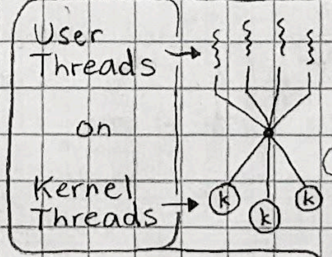
Kernel Thread Problems →
① Every thread operation must go through kernel.
② One-size-fits-all thread implementation.
③ Generally heavyweight memory requirements.

⑦ Threads, in-depth

User Threads

Problem: any block/page fault blocks all threads.

Implementation
- Allocate a new stack for each `pthread_create`.
- Keep a queue of runnable threads.
- Switch away from blocking system calls. (use non-blocking versions)
- Implement a scheduler (with `setitimer`)



Summary: Groups of User threads implemented on individual kernel threads.

AKA n:m threading (n → user #, m → kernel #)

Problems:
- Miscommunication between user-level and kernel.
- Similar issues as n:1 threads.
- Kernel doesn't know relative importance.

CS 350 Notes 2

8 Go Language + Concurrency

Go Routines
- Lightweight
- Segmented Stack
- Routines on top of kernel threads (aka user-level!)
- Converts blocking system calls

Go Channels
Communication between routines.

Allows synchronization.

aka SC

Critical Section: the part of a concurrent program that touches shared objects/variables.

Problem: without synchronization, data races occur.

Options: • Atomic instructions (instant). • Locks.

Sequential Consistency: • Maintain program order. • Ensure *write atomicity*. ★

Properties of Solution ★
1. Mutual Exclusion
2. Progress: can enter in a bounded time period
3. Progress vs. Bounded Waiting
   - ↳ smth always running. ↳ thread A eventually runs.

EG Peterson's Solution — SUBOPTIMAL
- → Just 2 threads, know their own ID
- → "wants" array, stores if thread i wants to run.

9 Consistency, Mutexes

x86 Consistency
- WB: (default) Write-back
- WT: Write-through
- UC: Uncachable
- WC: Write-combining

"If you obey certain rules, behaviour will be indistinguishable from SC."

x86 Atomicity
"lock" prefix forces atomicity.
↳ expensive; can avoid if mem is exclusively cached

"xchg" is always locked.

"cmpxchg" is always locked.

"lfence" can't reorder reads.

"sfence" can't reorder writes.

"mfence" can't reorder either.

xchg – exchange: test & lock in 1 atomic op.

Lock Implementation (with xchg) ★
```
Acquire(bool *lock) {
    while (xchg(true, lock) == true);
}
Release(bool *lock) {
    *lock = false;  // give up the lock
}
```
Known as a spin-lock. Has a problem!

Solutions: ① xchg in "Release" ② fence

Mutexes ★ (aka Locks)
```
mutex_lock(lock)
  // critical section
mutex_unlock(lock)
```
- → Builds on the spin lock paradigm
- → Blocks on failure instead of spinning

Thread API Contract ★
1. All global data is protected by a mutex
   - ↳ The responsibility of the application writer
2. If mutexes used properly, you should see SC.
3. The OS kernels need synchronization
   - ↳ Anywhere in the kernel with interrupts ruins mutexes.

10 Waiting + Threading APIs

Wait Channels ★ Blocking!
Wait channels implement thread blocking. Useful abstraction!

- void WaitChannel_Lock(WaitChannel *wc) → prevents races.
- void WaitChannel_Sleep(WaitChannel *wc) → blocks any calls to WC.
- void WaitChannel_WakeAll(WaitChannel *wc) → unblocks sleeping threads
- void WaitChannel_Wake(WaitChannel *wc) → unblocks 1 sleeping thread.

Producer <> Consumer ★
Producer creates an item on a buffer, for consumer to read from.

They use "global" counters to avoid overwriting previously produced items.

11 Mutexes, Condition vars, ~~Semaphores~~

PThread Mutex API ★
- init • lock • unlock
- destroy • trylock (0-success -1-error)

Condition Variables ★
Busy-waiting sucks; rather inform scheduler!

cond_t nonempty: used when wanting buffer not empty.

cond_t nonfull: used when wanting buffer not full.

⑫ Semaphores, Monitors, Hand-over-Hand, Benign Races

Semaphore ★

Synchronization primitive
- has a count!
- sem_wait() → count--;
- sem_post() → count++;

If N=1, the semaphore is a mutex! or: producer + consumer

Usage
- Limit # of threads that can reach a part of code

Implementing Mutex ★

(all are mutex locks)

```
-spinlock_lock()
while (lock is locked) {
  — waitchannel_lock()
  — spinlock_unlock()
  — waitchannel_unlock()
  — spinlock_lock()
}
- status = LOCKED
-spinlock_unlocked
```

Monitors ★

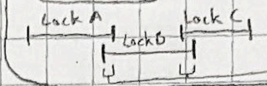
An abstraction: a region within your object where only one thread can go in at a time.

eg. "synchronized" keyword on an object method.

Conditions: (condition variables) obj.wait() obj.notify() Obj.notifyAll()

Hand-over-Hand Locking

EXAMPLE ★ Ensuring at least one lock is always acquired.



★ The idea is that overlap.

Benign Races

By "cheating," we can be faster! So if the race doesn't ruin the result, we can allow it to happen.

⑬ Data Races, Deadlocking

Detecting Data Races ★

① Static analysis: hard, incosistent
② Instrumentation: trap memory accesses
③ Lockset algorithm: set of locks
  ↳ kept for every memory location
  ↳ if no single lock is kept, warning!
④ Happens Before: lock <> memory access

Deadlock Problem

★ Don't acquire locks in different orders! ★
★ Don't have circular dependencies with condition variables ★
  ↳ Don't hold locks across abstraction layers (eg. function call)

Required conditions

① Mutual Exclusion (you need some locks)
② No Preemption (preemption may lead to a livelock tho)
③ Multiple Independent Requests
④ Circularity in Graph of Requests

⑭ OS Implementation

Multicore Caches ★

Caching across cores for performance
- Bus-based Approach
  ↳ "snoopy" protocols (listeners)
  ↳ Unfortunately, limitted scalability
- Networked Approach
  ↳ Divide cache into chunks (cache lines)

3-State Coherence Protocol (MSI)

Modified: only one cache has a valid copy.

Shared: more caches (and memory) are valid.

Invalid: doesn't contain any data.

Amdahl's Law

How long it takes to run a program using n cores instead of just 1.

$$T(n) = T(1)\left(B + \frac{1}{n}(1-B)\right)$$

$T(n)$: Time for n cores; B: fraction of job that must be serial

Core & Bus Actions ★

Read (load): cacheline enters "shared"

Write (store): invalidate all other cache kopies (& cacheline "shared")

Evict: writeback contents to memory if modified; discard if "shared".
  ↳ Used when no more room in the cache.

LESSONS for MULTITHREADING ★

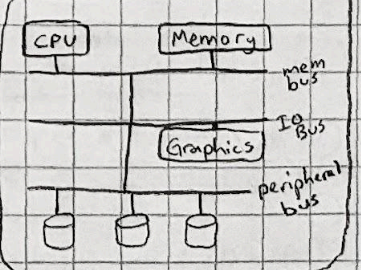
Avoid False Sharing: don't put data used by different threads in same cache line.

Align Structures: combine related data.

Pad Data Structures: blocks of 64 bytes.

Avoid Contending: reduce costly traffic between cores.

Memory & IO Busses



Accessing Devices

- Special I/O Instructions: transfers data between port and CPU register.
- Memory-mapped I/O: each device register has a physical memory address that drivers can use.

CS 350 Notes 3

15 Devices & I/O

Physical Disk Drive ★

Head: record/sense data along cylinders.
(usually only 1 head (w/circuitry))

Disk: Array of sectors (numbered)

Sector: Unit of transfer between disk & Memory.
ATOMIC

Memory-mapped I/O

Essentially hard-coded locations in memory where I/O will be.
Status field to know when I/O is ready.

Device Driver: A part of the kernel that interacts with a device.
→ Use interrupts to avoid polling

Direct Memory Access (DMA) ★

1. CPU initiates DMA request to device (eg. Disk)
2. Device directs data transfer
3. Device interrupts CPU on completion

Positioning & I/O

SEEK:
- speedup ← dominates short seeks
- coast (40 g of acceleration)
- slowdown
- settle ← dominates very short seeks (1ms)

Cost Model for I/O

Seek Time: move heads to appropriate cylinder.

Rotational Latency: rotate to correct sectors.

Transfer Time: wait while desired sectors spin past head(s)

16 Disk Scheduling

Disk Performance

- Try to achieve contiguous accesses
- Try to order requests for minimum seeks

Scheduling ★

"First Come First Serve" FCFS
→ Easy to code, good fairness
→ can't exploit request locality

"Shortest (Seek) Position Time First" SSTF
→ Exploits locality, higher throughput
→ Starvation (edges ignored)
→ Improve: AGED SPTF

"Elevator Scheduling" SCAN
→ Uses locality, bounded waiting
→ Middle is prioritized, might miss locality
→ Improve (Unix): CSCAN (circular scan)

"Varied Scan" VSCAN
It's SPTF but leaning towards SCAN
(other direction $= T_{pos} + r \cdot T_{max}$) ($r \approx 0.2$ is good)

17 Midterm Review Session

18 File Systems

What File Systems Do

- Don't go away (ever)
- Associate bytes with names (files)
- Associate names with each other (directory)

Files ★

→ Persistent, named data objects
→ May change size
→ Has associated metadata.

File Interface ★

- **open** returns a file descriptor
- **close** invalidates a file descriptor
- **read / write** copies data file ⇌ virtual address space
- **lseek** enable non-sequential read/write
- **get/set metadata** fstat, chmod etc

Flash Memory (SSD)

Flash Mem: Floating gate transistors

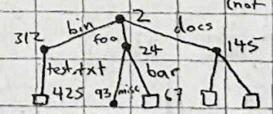
Data Arrangement ★

→ divided into Blocks & Pages
→ read/write at Page level
→ 1 → 0 is easy, but 0 → 1 takes high voltage, so you need to "flash" at block level.
→ We call this issue "Write Amplification."
→ Limited # of flashes
→ Solve via Translation Layer
- Allocate new virtual page if needed.
- Mark invalid, use garbage collection.

→ Ensure flashes are evenly distributed so that Wear Levels are optimized.

Directories & Filenames ★

A directory maps file names to i-number.
◉ i-number = unique identifier for a file (or directory)
Applications refer to files using pathnames (not i-numbers)



Eg: foo/bar → 67

19 More File System Stuff

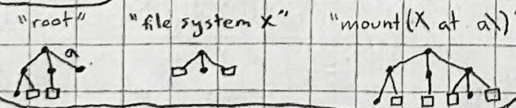
File system: manages your files! Example: C:

Mounting ★

While windows has two-part file names:
file system, pathname
C:\users\cs350\urmom.txt

Unix creates a single hierarchical namespace that combines both namespaces.

"mount" → doesn't turn 2 file systems into 1.



Links ★

hard link: association between name and i-number.
→ Each entry in a directory is a hard link.
[it's not possible to link to a directory]

- **link** link path to file (not directory)
- **unlink** removes hard link to path

i-node

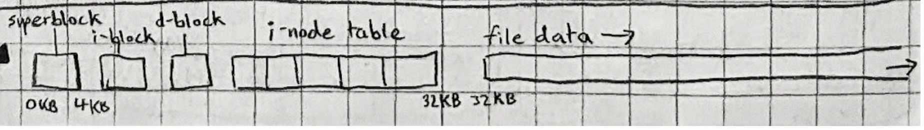
holds all metadata!

file type, permissions, file length, # of file blocks, last access time, last i-node/file update, hard links #, direct data block pointers, indirect data block pntrs

Indirect Pointer: pointer to block of direct pointers

File System Implementation ★

1. **Store:** data, metadata, directories & links, filesystem metadata
2. **Non-persistent store:** open files per process, file position per open file, cache
3. **Group Sectors ⇒ Blocks:** better spatial locality, fewer block pointers
4. **Allocate your memory:** A) Most for data B) i-nodes C) unused i-nodes + data (bitmap) D) file sys metadata



(20) File system stuff (again)

Let's talk exam questions! Examples:
- Max file size of a file system?
- Why things are done a certain way?
- You wanna access a particular offset of a file: how do you get there?

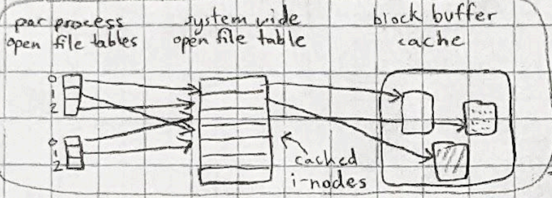
File System Design ★
- most common size: 2K
- average file size: 200K
- most bytes are in large files
- system is roughly half full
- directories are small: ≤20

Reading from a file (/foo/bar) ★
1. i-node for root directory (assuming 1 data block)
2. Check if foo exists (get i-number)
3. Read foo's i-node
4. Read foo's data blocks → bar's i-number
5. Read bar's i-node (potentially cached! ◡̈)
6. Read bar's data blocks!

In-Memory (Non-Persistent) Structures
Open File Tables: per process & system wide

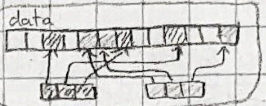
The process's file table points to entries in the system-wide file descriptor table. If multiple processes open same file, they'll point to different spots in system-wide file table, unless it was a fork!

- Cache: i-nodes, data blocks, indirect blocks.

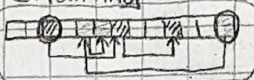


Other File Systems
our way:

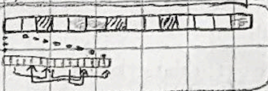
Per-file indexing



Chaining



External chaining

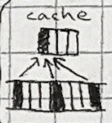


Creating a file (/foo/bar) ★
1. read i-node from root → foo exists?
2. foo's i-node → read foo's data → bar doesn't exist
3. allocate an i-node, update i-node bitmap
4. add a hardlink to foo for bar
5. read whole i-node list, update foo's, write back!
6. update foo filesize to finish.

(22-23) Cache & Paging (CS 251)
★ It is certain there will be a cache or virtual memory question on the exam (so review CS251)

Caching ★
Block: minimum information unit size

Hit/Miss: whether info is present

Direct-mapped Cache


To differentiate between potential OG locations:
→ Store a "tag" with the data.

N-way Set-associative cache

Tag		index	block			byte		
63		…	5	4	3	2	1	0

A 2^n-way cache of size 2^k bytes with a 2^b block size: (block size $2^3 = 8$)
- line size = bytes + blocks = $2 + b = ls$
- index bits = $i = k - 2 - b - n = k - ls - n$
- tag bits remain: $64 - ls - i$ = tag

(21) Failures, Virtual Memory

Failures ★
Idea: imagine the computer crashes during one of your file-system-related system calls.

Solution: ensure all persistent structures are crash consistent.

Fault Tolerance
Idea: special purpose consistency checker: runs after crash, looking for inconsistencies:
- eg. file with no directory entry
- eg. free space that's not marked free

Journalling: implement write-ahead logging of filesystem meta-data changes. Afterwards, write "commit" and go actually do it.

Virtual Memory ★
Purpose:
- Processes believe they're alone
- Functional isolation between processes.

Summary:
- Running apps only see virtual addresses.
  ↳ including program counter, branches etc.
- A process's VM holds the code, data, and stack for the program it's running.

Address Translation
- The map between physical & virtual.
- Performed by the hardware.

Dynamic Relocation
The CPU has an MMU (memory management unit) with:
→ relocation register (R)
→ limit register (L)

The only thing the OS does is update the R and L values whenever you context switch or update physical space.

Paging ★ TLB? "Translation lookaside buffer" (cache for Page Tables)
Virtual memories are divided into fixed-size chunks called pages. Page size = Frame size.

Page Tables
Each process has a page table.

	Page#	Frame#	Valid?
PTE →	0x0	0x27	1
	0x1	0x1c	0
	⋮	⋮	⋮

↳ other PTE fields include: "protection bit," "reference bit," "dirty bit," etc.

⇒ Address Translation:
1. Page # = Virtual Address / Page size
2. Offset = Virtual Address % Page size
3. Physical Address = Frame # · Frame size + offset (only if the Valid bit is 1)

Segmented Address Space
Provide a separate mapping for each segment of a virtual address space.
→ Now virtual addresses have 2 parts: ① segment ID, ② offset in segment

Two-level Paging
Like indirect pointers but for Page Tables.

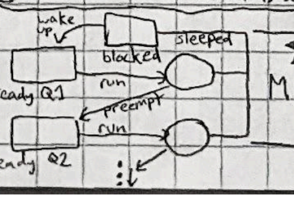
Secondary Storage
Exploit secondary storage (ie. disks or SSD) by letting VMs use them, which introduces:

PTE's "present" bit which says if page is in memory (1) or just in secondary (0).

VM for Kernel
To make Kernel in VM:
1. Sharing: share data with other programs by making the kernel's VM overlap with the processes' VM.
2. Bootstrapping: what about when it's starting?? (architecture-specific)

(23) Final VM (more CS 251) & Scheduling

Page Replacement (CS 251)
1. FIFO – bad. ◠̈
2. LRU (Least-Recently Used) – good. ◡̈
3. Clock – FIFO unless use bit is set.



Scheduling Models
- FCFS – first come first serve
- SJF – shortest job first
- SRJF – SJF w/ pre-emption
- RR – Round Robin
- MLFQ – Multi-level Feedback Queue.
  ↳ A priority-Queue of RRs
  - move up: if process gives back control.
  - move down: if process is greedy.

↑ bernard carrying ~~me~~ my OS skills ↑

↑ unsuccessfully hiding from midterm grading ↑

↓ us running from the final after completely destroying it ↓